
# Neural Network Classification of Environmental Samples

THESIS

Jeffrey L. Blackmon
1st Lieutenant, USAF

AFIT/GEE/ENG/96D-04



DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GEE/ENG/96D-04

# Neural Network Classification of Environmental Samples

THESIS

Jeffrey L. Blackmon
1st Lieutenant, USAF

AFIT/GEE/ENG/96D-04

AFIT/GEE/ENG/96D-04

# Neural Network Classification of Environmental Samples

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Engineering and Environmental Management

Jeffrey L. Blackmon, BSEE

1st Lieutenant, USAF

December, 1996

*Acknowledgements*

In completing the course work and pursuing the research necessary to complete graduate school at AFIT, numerous individuals have been critically important, especially concerning the completion of this research and the accompanying document.

My utmost respect and appreciation go to Dr. Steven K. Rogers. Working with Dr. Rogers has been one of the most enjoyable aspects of this research effort. He is truly unique among the instructors at AFIT. All of the students whom he advises have access to an intellect of unmatched breadth and depth. His expertise have been essential in this research effort.

Further, I would like to thank the other members of my committee, Lieutenant Colonel Michael Shelly and Major Jeff Martin. They have been more than willing to provide instruction and guidance throughout the course of this effort.

Often overlooked in the daily grind of graduate school, is the assistance provided by fellow students. AFIT students are always willing to lend a hand even in their own "crunch" time. Though the students are numerous, I would like to give a special thanks to Captain Anthony Ference, USMC, who on many occasions put red ink to my not so carefully crafted documents. I would also like to thank all of those students who work in the Hawkeye lab daily and whom I have troubled on many an occasion with various research related questions.

Finally, and certainly not least by any means, my most heart-felt appreciation goes to my loving wife Allison. She possesses those intangible qualities which are critical to all of my endeavors. She possesses more compassion, generosity, and capacity for unconditional love than anyone that I have ever known. These qualities in her create a base of support from which no goal is unattainable.

Jeffrey L. Blackmon

*Table of Contents*

*List of Figures*

*List of Tables*

*Abstract*

This research develops a general methodology for designing neural network classifiers for real-world environmental problems. This methodology is demonstrated through the design of a multi-layer perceptron to classify stainless steel and actinide samples. This research provides techniques for selecting architecture and training parameters, choosing the number of training epochs, reducing the feature sets, and evaluating classifier performance. For the stainless steel data, the feature set is reduced from 196 features to 54 features and the average training set error and test set error are .6% and 5.5%, respectively. The best results attained on the actinide data set are 24% training set error and 26% test set error. The actinide feature set is reduced from 18 feature to 9 features. The products of this effort are a concise methodology for developing neural network classifiers and the specifications for a multi-layer perceptron classifier for each of the data sets.

Neural Network Classification of Environmental Samples

# *I. Introduction*

## *1.1 Background*

Like many Air Force organizations, the Air Force Technical Applications Center (AFTAC) has a mission, insuring Nuclear Test Ban Treaty compliance, which involves monitoring of the environment. This environmental monitoring often requires identification of unknown chemical compounds and, for AFTAC, often requires determining the origin of environmental samples.

The traditional method used to identify environmental samples is to determine the individual components of the sample, thereby elucidating the identity of the sample itself. The field of analytical chemistry has conceived numerous instrumental methods to identify the constituents of a sample including mass spectrometry, raman spectroscopy, fluorescence spectroscopy, and gas chromatography [16]. Each of these methods produces spectra from which the relative quantities of the sample's constituents may be determined. Consequently, these quantities may be used to classify the sample by comparison to those of known samples. However, the task of classifying a sample given these quantities is often complicated by similarity of samples and the large number of known chemical compounds. Consequently, automated methods of classifying environmental samples are necessary.

Neural networks, sometimes called artificial neural networks, have been shown capable of classifying complex patterns such as those mentioned above [4]. Artificial neural networks are physiologically motivated computer algorithms which attempt to mimic the function of the large interconnected network of neurons in the human brain, which has extraordinary pattern recognition capabilities[23]. These artificial neural networks *learn* to map a set of input features, elemental composition, onto a set of outputs such as a binary node whose output (1 or 0) represents *steel* or *not steel.* For this reason, neural networks may be used to classify the given environmental data.

Numerous efforts have illustrated the use of neural networks for classifying chemical spectra. A thorough review of these efforts is provided by Burns and Whitesides [4]. However, the efforts focus primarily on the chemical analyses and the neural network results, rather than the methodology for designing the neural network classifiers. In addition, classifiers are unique to the data being classified and must be designed specifically for the given data. Furthermore, all efforts thus far have used the analytical spectra as input features rather than the relative quantities which may be derived from the spectra, and no effort has been made to correlate samples to the location from which they were taken. As a result, this thesis focuses on classification by name and by location using elemental percentages (by weight) and radioactive particle percentages (by atom) as input features.

### *1.2 Problem Statement*

A methodology for the design of neural network classifiers for environmental applications is developed and is demonstrated through the classification of stainless steel and actinide samples given the relative quantities of the constituent elements and particles of each sample.

### *1.3 Scope*

Elemental percentage by weight and radioactive particle percentage by atom for the stainless steel samples and radioactive particle percentages alone for the actinide samples have been provided by the Air Force Technical Applications Center. A neural network is designed to classify this data and the salient features (features most useful in classification) are determined. Once designed, the performance of the classifiers is evaluated by estimating the Bayes error bounds. While the results obtained herein are unique to this particular data set, the methodology is general enough to be applied to other environmental data sets.

### *1.4 Research Objectives*

The research objectives for this thesis are as follows:

1. Provide a methodology for developing neural network classifiers for environmental applications.
2. Demonstrate the ability of neural networks to classify real-world environmental data.
3. Design a neural network to classify the given data sets.
4. Determine the salient features for the given data sets.
5. Evaluate the performance of each classifier.

### *1.5 Approach*

The approach taken in this effort consists of several distinct steps. Initially, the data is preprocessed. This preprocessing consists of several steps including normalizing the data. The second step is to develop a multi-layer perceptron for classifying the stainless steel and actinide data sets. This includes selection of architecture and parameter values as well as training and testing the classifier to achieve an acceptable level of performance. Following testing of the multi-layer perceptron, a technique known as forward sequential selection is used to rank the input features in the order of decreasing importance to classification. Then, another classifier is trained and tested using subsets of the input features in order to enhance the performance of the classifier. Finally, the performance is evaluated by comparing the classifier error to Bayes error bounds.

### *1.6 Thesis Overview*

The remainder of this thesis is organized as follows: Chapter II provides an overview of the neural networks employed in this research. Chapter III describes the implementation and evaluation of these networks and presents the results of this effort. Finally, a summary of these results and conclusions are presented in Chapter IV.

# *II. Theory*

## *2.1 Introduction*

The purpose of this chapter is to provide an introduction to pattern recognition and neural networks, and to overview the theory necessary to understand the methods presented in Chapter III. Only concepts relevant to this research are covered in this chapter. The topics covered in the following sections are:

- Introduction to Pattern Recognition
- The Single Perceptron
- The Multi-Layer Perceptron
- Feature Selection
- Bayes Error Bounding

## *2.2 Introduction to Pattern Recognition*

According to Duda and Hart, pattern recognition is the assignment of an object to any of several categories based on measurements of the object's features. More specifically, pattern recognition is selecting the actual state of nature $\omega$ of an object from a set of possible states $\omega_i$ based on some measurement $x$ [9].

The process of selecting the class of an object based on some analog measurement of that object amounts to drawing a decision boundary in the feature space which separates the members of competing classes. For example, suppose that indoor air samples are to be classified as *hazardous* or *non-hazardous* based on average particle size $x_1$. If a number of samples for which the class memberships are known are plotted in the feature space, the optimal decision boundary is the line which absolutely separates the two classes as shown in Figure 2.1a. If, however, the samples are to be classified based on both average particle size $x_1$ and radon concentration $x_2$, the decision boundary must be drawn in a two-dimensional feature space as shown in Figure 2.1b.

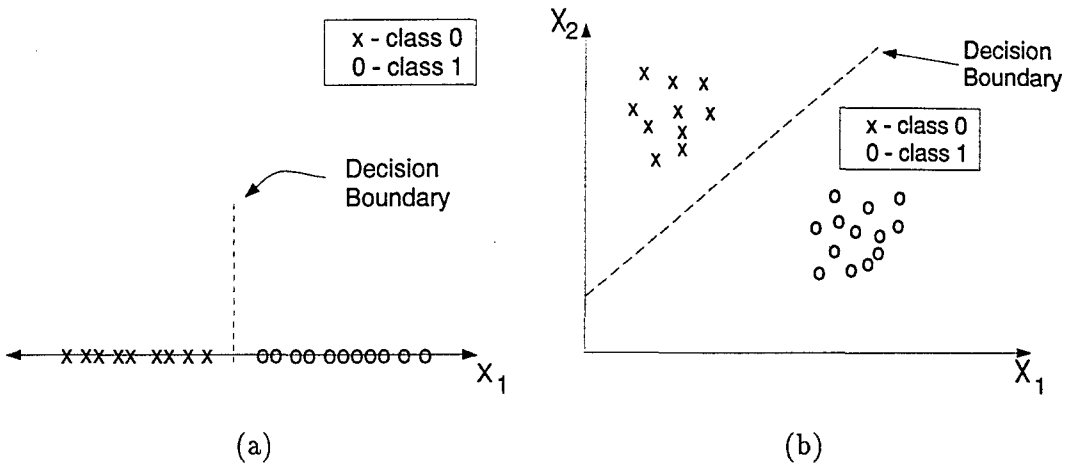



Figure 2.1 Two-class classification problem: (a) one-dimensional case (b) two-dimensional case.

Once the decision boundary has been set, any subsequent samples of unknown class membership are then classified according to their position in the feature space relative to the decision boundary.

This is a grossly oversimplified representation of real-world pattern recognition problems for several reasons. Classification of objects will usually be based on multiple features. In general, $x$ will be a vector containing measurements of multiple features. In this thesis for example, each sample is represented by a vector containing the percentages of each periodic element and the percentages of certain radioisotopes which compose the sample. Furthermore, it may be impossible to draw a decision boundary which completely separates the class data. In reality, the boundary is drawn such that the probability of classification error is minimized. This concept is discussed further in section 2.6. The boundary may also be more complex (consisting of multiple lines or non-linear boundaries) in many cases such as non-linearly separable problems and multi-class problems. The classic example of non-linearly separable data is the XOR problem shown in Figure 2.2a. Clearly, the two classes of data cannot be separated by a single line rather it requires a complex decision boundary as shown. Similarly, Figure 2.2b shows a three-class problem which also requires a complex boundary. As the dimensionality (the number of features) of a classification problem or the number of classes grow, the decision boundary becomes even more complex

consisting of surfaces, rather than lines, in the feature space. In this case, sophisticated algorithms must be used to derive decision boundaries.

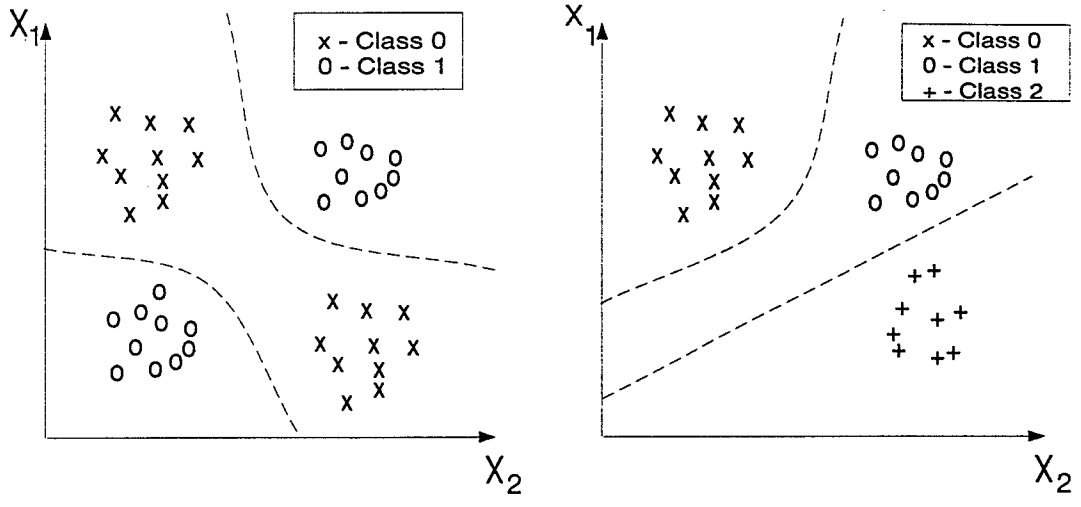



Figure 2.2 Complex decision boundaries: (a) XOR data (b) three-class data.

Automated pattern classification may be accomplished using machines or algorithms known as pattern classifiers, or simply classifiers. Further, automated pattern recognition consists of two distinct stages: determining the decision boundaries based on existing data (training the classifier), and classifying new data (testing the classifier). Numerous statistical classifiers exist which utilize the probability distributions associated with the state of nature $\omega_i$, $p(\omega_i)$, and the measurement $x$, $p(x)$, and the conditional probability distributions $p(\omega_i|x)$ to determine the decision boundaries. An overview of Bayes Decision Rule, which is the fundamental basis of statistical classifiers, is provided in section 2.6. For a thorough treatment of statistical classification methods, the reader is referred to Duda and Hart [9].

Artificial neural networks may also be employed for pattern classification problems. These classifiers were precipitated by the realization that animals, especially humans, have the ability to rapidly classify complex patterns. This led to much research in the 1950's and 1960's out of which came mathematical models which were intended to elucidate the function of the human brain (an overview of these research efforts is given by Rogers *et*

*al* [23] and by Lippmann [17]). The value of these models as problem-solving machines soon became apparent and the study of pattern recognition branched into two subfields, the study of human and animal pattern recognition and the development of mathematical analogs of biological systems capable of pattern recognition [31]. The latter has become a large field concerned with the development of artificial neural network architectures and learning/training algorithms.

Although many different architectures and algorithms have been developed [15, 6, 12], the most popular is the multi-layer perceptron and its various training algorithms. This popularity is due to its ease of implementation and its ability to represent any decision boundary[8]. Furthermore, the multi-layer perceptron is a Bayes optimal classifier (see Section 2.6), which means it cannot be outperformed, on average, by other types of classifiers on the same data set [26]. For these reasons, the multi-layer perceptron is used throughout this research.

## *2.3 The Perceptron*

The perceptron, which was introduced by Rosenblatt in 1959 [24], is the simplest neural network and is the basic functional unit of the more complex multi-layer perceptron that will be used in Chapter III. The single perceptron may be used to solve two-class problems in which the class data is linearly separable. This section covers the architecture and operation of the single perceptron and provides an overview of the perceptron training algorithm.

*2.3.1 Architecture and Operation.* The function of the perceptron (Figure 2.3) is analogous to that of the biological neuron [23]. The perceptron computes a weighted sum of its inputs.

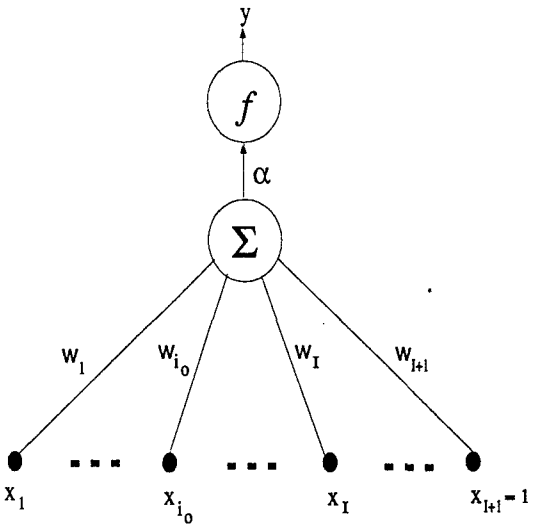



Figure 2.3 Single perceptron.

More specifically, the perceptron multiplies each of its inputs, the input features $x_0 \ldots x_I$ and a bias $x_{I+1}$, by its respective weight $w_i$ and computes the sum. The sum may then be transformed by an activation function, which forces the output between a high value (typically 1) and a low value (typically -1 or 0). The bias is typically set equal to 1 resulting in a bias term in the sum which is simply the weight $w_{I+1}$. This calculation may be represented mathematically as

$$y = f(\sum_{i=1}^{I} w_i x_i + w_{I+1}) \tag{2.1}$$

or

$$y = f(\sum_{i=1}^{I+1} w_i x_i) \tag{2.2}$$

where $f(\bullet)$ is the activation function and $I + 1$ is the number of input features including the bias value of 1.

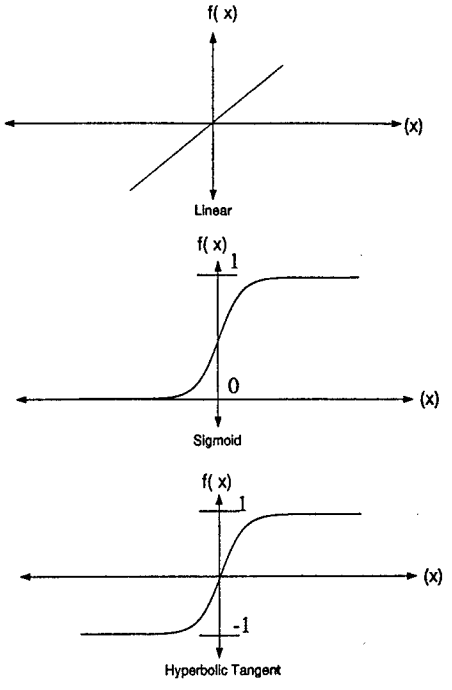



Figure 2.4 Activation functions.

While any number of functions may be used, the activation functions typically associated with the single perceptron are shown in Figure 2.4. The mathematical formulations of these functions are as follows:

$$\text{linear: } f(x) = x$$

$$\text{sigmoid: } f(x) = \frac{1}{1+e^{-x}}$$

$$\text{hyperbolic tangent: } f(x) = tanh(x)$$

*2.3.2 Training.* The single perceptron may be trained to classify input vectors in a two-class problem. This training is accomplished by randomly initializing the weights to some small value (usually in the range [-0.5,0.5]), presenting class-labeled input vectors to the perceptron and adjusting the weights such that the output of the perceptron tends toward 1 for members of class $\omega_1$ and toward 0 or -1, depending on the activation function, for members of class $\omega_2$. The error between the perceptron output and the desired perceptron output is used to accomplish this weight adjustment. Mathematically this is

represented by

$$\mathbf{w}^{+} = \mathbf{w}^{-} - \eta \frac{\partial \mathbf{E}}{\partial \mathbf{w}} \tag{2.3}$$

where $\mathbf{w}^{+}$ represents the adjusted weight vector, $\mathbf{w}^{-}$ is the previous weight vector, $\eta$ is a weight adjustment factor commonly referred to as the learning rate, and $\mathbf{E}$ is the error. Derivation of the functional form of the weight update is done only for the multi-layer perceptron (Appendix A). The learning rate is a value in the range (0,1). All of the vectors in the training data set are repeatedly presented to the network, and the weights are adjusted until a suitable level of error is reached. Each pass through the training data set is known as an epoch.

Although the perceptron is easy to implement and train using this algorithm, it is limited to two-class problems in which the data are linearly separable. To overcome this limitation, a more complex structure must be implemented, the multi-layer perceptron.

## *2.4 The Multi-Layer Perceptron*

The multi-layer perceptron is the most frequently implemented of all neural network architectures. Unlike the single perceptron which only draws a single line or plane in the feature space, the multi-layer perceptron is capable of drawing complex surfaces to separate class data. The remainder of this section provides an overview of the architecture, operation, and training of the multi-layer perceptron.

*2.4.1 Architecture and Operation.* The multi-layer perceptron is, as the name implies, nothing more than two or more layers of perceptrons as shown in Figure 2.5. Although multi-layer perceptrons may consist of more than two layers, it has been shown that two layers are sufficient for any arbitrary classification problem[8].

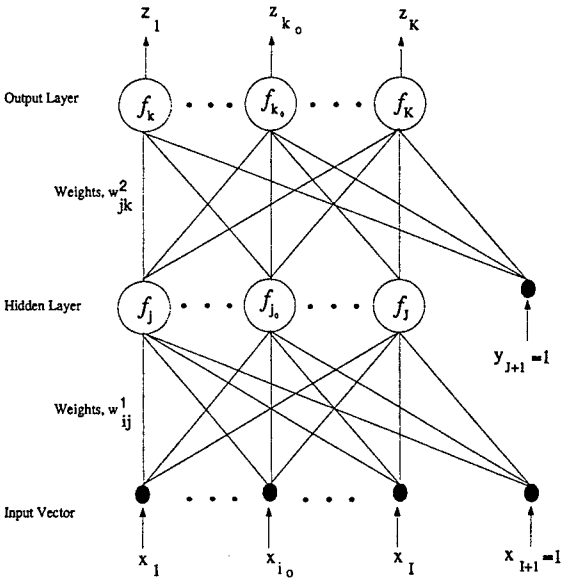



Figure 2.5 Multi-layer perceptron.

Each perceptron, or node, in the first layer, often referred to as the hidden layer, computes a weighted sum of the inputs (as described in Section 2.3.1) using its own weight vector. The weight vectors of the hidden layer nodes comprise the weight matrix $\mathbf{w^1}$. This operation is represented by

$$y_j = f_j(\sum_{i=1}^{I+1} w_{ij}^1 x_i) \tag{2.4}$$

or, in matrix/vector notation,

$$\mathbf{y} = \mathbf{f_j}(\mathbf{w^1 x'}) \tag{2.5}$$

where $\mathbf{x}$' is the transpose of $\mathbf{x}$, and $\mathbf{f_j}$ is the transformation function for the hidden layer nodes. Similarly, each output layer node computes a weighted sum of the hidden layer outputs and a bias term, $y_1 \ldots y_{J+1}$, producing the multi-layer perceptron output vector $\mathbf{z}$. The weight vectors for the output layer nodes are given by $\mathbf{w^2}$. This operation is represented by

$$z_k = f_k(\sum_{j=1}^{J+1} w_{jk}^2 y_j) \tag{2.6}$$

or, in matrix/vector notation,

$$\mathbf{z} = \mathbf{f_k}(\mathbf{w^2 y'}) \tag{2.7}$$

where $\mathbf{y}$' is the transpose of $\mathbf{y}$, and $\mathbf{f_k}$ is the transformation function for the output layer nodes.

The activation functions used with the single perceptron are also used in multi-layer architectures with all nodes in a layer employing the same function. For example, all hidden layer nodes may use the sigmoid function, while all output layer nodes use a linear transformation, sigmoid-linear. Although many different combinations may be used, common combinations of activation functions are sigmoid-sigmoid, sigmoid-linear, tanh-tanh, tanh-linear.

In designing a multi-layer perceptron for a particular problem, several parameters must be chosen. Among them are the number of hidden layer nodes ($J$), the number of output layer nodes $K$, and the activation functions to be employed. While $K$ is usually set to the number of classes in the problem so that each node represents a class, $J$ is somewhat arbitrary. A common rule of thumb is that the number of hidden layer nodes should be selected such that the number of samples is at least 10 times the number of weights in the network [2]. In addition, the activation functions must also be selected. Selection of architecture parameters for this research are covered in Chapter III.

*2.4.2 Training.* As with the single perceptron, training is accomplished by initializing the weight matrices to some small value, presenting the class labeled input vectors, and adjusting the weight matrices until a suitable level of error is reached.

The multi-layer perceptron is most often trained using the backpropagation algorithm which was independently discovered by three different researchers [32, 20, 28]. Although numerous efforts have been made to improve the backpropagation algorithm(backpropagation with momentum[28], conjugate gradient[29], and newton's method[3], to name a few), the addition of momentum is the least difficult improvement to implement and is, therefore, the most common. The complexity of the latter two algorithms and their intense computation time often eclipse their improvement in generalization (ability to classify data which was not used in training) and convergence time (number of epochs required to reach an acceptable level of error)[1]. For this reason, backpropagation with momentum is used to train the multi-layer perceptrons implemented throughout this research.

Backpropagation adjusts the weights based on the error between the desired output of the network and the actual output for each input vector in the training data set. The most common measure of this error is the sum-squared error which is given by

$$E = \frac{1}{2}\sum_{k=1}^{K}(d_k - z_k)^2, \tag{2.8}$$

where $d_k$ is the desired output, and $z_k$ is the actual output.

Training may be conducted in one of two modes, instantaneous or batch. In instantaneous training, the weights are adjusted after the presentation of each input vector. The general weight update rule for such training is

$$\mathbf{W}^+ = \mathbf{W}^- - \eta\frac{\partial E}{\partial \mathbf{W}^-} \tag{2.9}$$

where $\mathbf{W}^+$ is the updated weight matrix, $\mathbf{W}^-$ is the old weight matrix, and $\eta$ is the learning rate. The learning rate is a variable which may assume values between zero and one. The learning rate may remain constant throughout training or it may be varied in this range during training. Algorithms which vary the training parameters during training are known as adaptive algorithms [7]. When performing batch training, each input vector is presented and the update portion of the equation is computed for each input vector. However, the weights are not updated after each input vector presentation. An average update is computed over all of the input vectors, and the weights are updated at the end of the epoch. In batch mode, the general update rule becomes

$$\mathbf{W}^+ = \mathbf{W}^- - \frac{1}{n}\sum_{i=1}^{n}\eta\frac{\partial E}{\partial \mathbf{W}^-} \tag{2.10}$$

where $n$ is the number of input vectors. For ease of notation, the remainder of the weight update equations in this chapter are shown in their instantaneous form. These equations may be transformed to their batch form by simply averaging the update portion of the equation over all of the input vectors.

In either mode, backpropagation training is simply a gradient descent which seeks the minimum in the weight space error surface by adjusting the weights in a negative gradient

direction. Figure 2.6 shows an error surface in a one-dimensional weight space, while Figure 2.7 illustrates an error surface in a two dimensional weight space. Note that as the number of weights increases beyond two, which is the case for any problem of significance, the error surface becomes impossible to visualize.

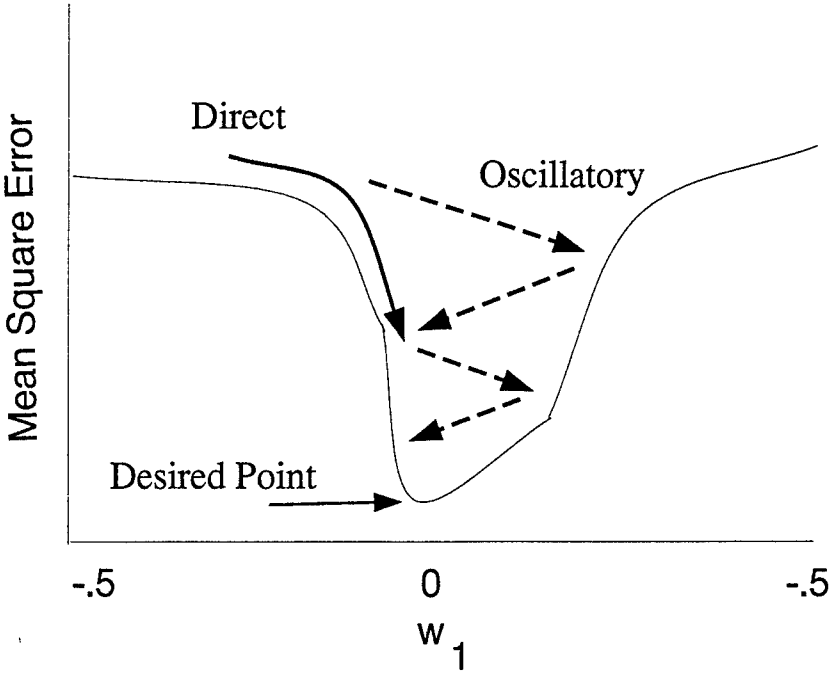



Figure 2.6 Error surface in one-dimensional weight space.

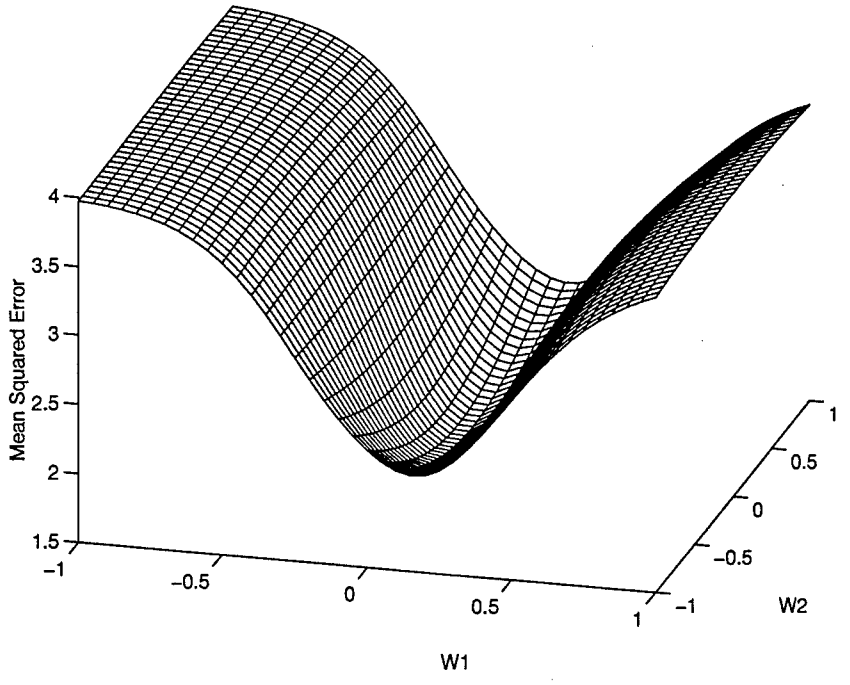



Figure 2.7 Error surface in two-dimensional weight space.

It is desirable for the algorithm to adjust the weights such that the most direct path to the error surface minimum is taken. However, the error surface descent is dependent on the learning rate $\eta$. As a result, the algorithm may skip back and forth over the minimum, increasing the time required to converge. This oscillation may be dampened by adding a momentum term which is a constant $\alpha$ multiplied by the previous weight update $\Delta W$. The momentum may be constant during training or may be adapted during training like the learning rate.

$$\mathbf{W}^{+} = \mathbf{W}^{-} - \eta \frac{\partial E}{\partial \mathbf{W}^{-}} + \alpha \Delta \mathbf{W} \tag{2.11}$$

This has the effect of increasing the weight update when moving down the error surface or decreasing the update when moving up the error surface. This improves backpropagation learning by allowing a more smooth and timely descent on the error surface.

The complexity of the backpropagation algorithm comes in deriving the derivative of the error with respect to the weights for the hidden layer and the output layer to arrive at a functional form of the weight update rule. The generic, functional form of the output

layer weight update is

$$w^{+}_{j_0 k_0} = w^{-}_{j_0 k_0} + \eta(d_{k_0} - f_{k_0}) f'_{k_0} f_{j_0} \tag{2.12}$$

, while the hidden layer update is given by

$$w^{+}_{i_0 j_0} = w^{-}_{i_0 j_0} + \eta \sum_{k=1}^{K} (d_k - f_k) f'_k w_{j_0 k} f'_{j_0} x_{i_0} \tag{2.13}$$

Note that the outputs of the hidden layer and the output layer are denoted by the transformation functions $f_j$ and $f_k$ respectively; $f'_{k_0}$ is the derivative of the activation function evaluated at node $k_0$. The general derivations of the hidden layer and output layer updates are given in Appendix A along with the transformation function specific derivations. The momentum term (not shown here or in the derivations) is simply the difference between the weight matrix from the previous input presentation and the current weight matrix multiplied by $\alpha$.

### *2.5 Feature Selection*

One of the most important tasks in classifier design is selecting the appropriate feature set for classification. This is critical for several reasons. According to the "Curse of Dimensionality" posited by Foley, a larger number of input features requires a larger number of training samples [10]. In addition, a larger feature set increases the number of weights that must be used when employing neural network classifiers. Both a larger training set and an increased number of weights lead to longer training times. Finally, Kabrisky has suggested that there is an optimal number of features in terms of error for some data sets [13]. Ultimately, the goal of feature selection is to reduce the feature set without a significant decrease, if any, in the accuracy. Many techniques exist for determining the saliency (importance for classification) of input features [27, 21, 30, 21] and, subsequently, for reducing the feature set. The method used for feature reduction in this research is forward sequential selection.

Forward sequential selection is commonly used to rank features according to their saliency. This techniques begins by training a classifier for each input feature from the set of candidate features. The feature which results in the highest classification accuracy is

added to a feature nucleus which is initially empty, and the feature is removed from the set of candidate features. Subsequently, a classifier is trained using the nucleus and each feature from the candidate set. Again, the feature whose addition to the nucleus results in the best classification accuracy is added to the nucleus and removed from the candidate set. This process is repeated until the desired number of features have been ranked or the candidate feature set is empty.

## *2.6 Introduction to Bayes Error Bounding*

Bayes error is the minimum average probability of error associated with any classification problem [9]. This means that Bayes error represents the best performance that a classifier may achieve on average. As such, it is the benchmark of performance for statistical classifiers, and any classifier with an error rate approaching Bayes error is considered to be Bayes optimal. Consequently, Ruck *et al* have shown that a multi-layer perceptron trained using sum-squared error as the error measurement approximates Bayes discrimination [27]. Thus, Bayes error is also used to evaluate the performance of the multi-layer perceptron classifier.

The remainder of this section discusses Bayes Decision Theory to the extent necessary to understand Bayes error and describes methods for estimating Bayes error.

*2.6.1 Bayes Decision Theory.* The stated purpose of pattern classifiers is to determine objects' classes, $\omega$, based on some feature or set of features, $x$, such that the probability of misclassification (error) is minimum. According to Bayes Decision Theory, the minimum probability of error is achieved by selecting the class $\omega_i$ which has the highest probability given the measurement $x$, $P(\omega_i|x)$. According to Bayes theorem, the *a posteriori* probability $P(\omega_i|x)$ is given by

$$P(\omega_i|x) = \frac{p(x|\omega_i)P(\omega_i)}{p(x)} \tag{2.14}$$

where $p(x|\omega_i)$ is class conditional probability of $x$ , $P(\omega_i)$ is the *a priori* probability of class $\omega_i$, and $p(x)$ is the probability density function (*pdf*) for the measurement $x$. The class conditional probability of $x$ is the probability of observing values of $x$ given a specific class

$\omega_i$ and the *a priori* probability of class $\omega_i$ represents the proportion of the total number of objects in the sample set which belong to class $\omega_i$. As an example consider a two class problem in one dimension. Bayes decision rule becomes

$$\text{select } \omega_1 \text{ if } P(\omega_1|x) > P(\omega_2|x) \text{ otherwise select } \omega_2$$

As shown in Figure 2.8a, Bayes decision rule produces a boundary at the intersection of the two distribution which results in the area under the two curves (the error) being minimized. Clearly, any other boundary will produce higher probability of error as shown in Figure 2.8b.

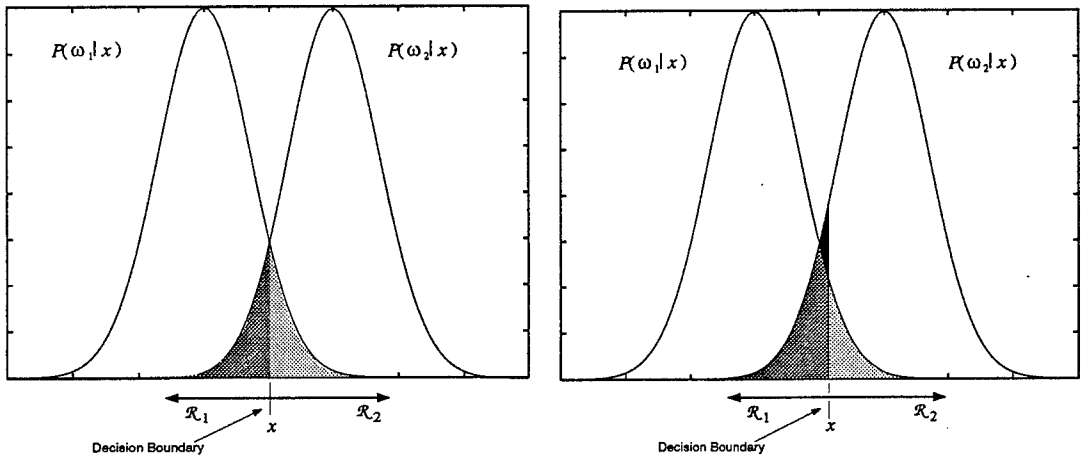



Figure 2.8 Probability distributions with decision boundaries: (a) optimal boundary set by Bayes decision rule (b) non-optimal boundary.

As stated, there is some probability of error associated with selecting the class $\omega_i$ given $x$. Because error is define as choosing the wrong state of nature $\omega$, the probability of error is the sum of the *a posteriori* probabilities of the states of nature not chosen. For example in a two-class problem, the probability of error given $x$ is given by

$$\mathrm{P}(error \mid x) = \begin{cases} \mathrm{P}(\omega_1 \mid x), & \text{if } \omega_2 \text{ is chosen;} \\ \mathrm{P}(\omega_2 \mid x), & \text{if } \omega_1 \text{ is chosen.} \end{cases}$$

The average probability of error is calculated over the entire range of $x$ as follows:

$$P(error) = \int_{-\infty}^{+\infty} P(error|x)p(x)dx \tag{2.15}$$

This represents Bayes error when the decision boundary is chosen according to Bayes decision rule.

*2.6.2 Bounding Bayes Error.* Because the sample set for any classification problem is finite and the underlying probability distributions are not often known, Bayes error cannot be calculated using the mathematical formulations given and may be estimated by calculating upper and lower bounds on the error. Any classifier whose error rate falls within these bounds is considered to be at optimal performance for the given problem. The most common error bounding technique employs the Leave-One-Out method to estimate the upper bound and the Resubstitution method to estimate the lower bound [11, 18].

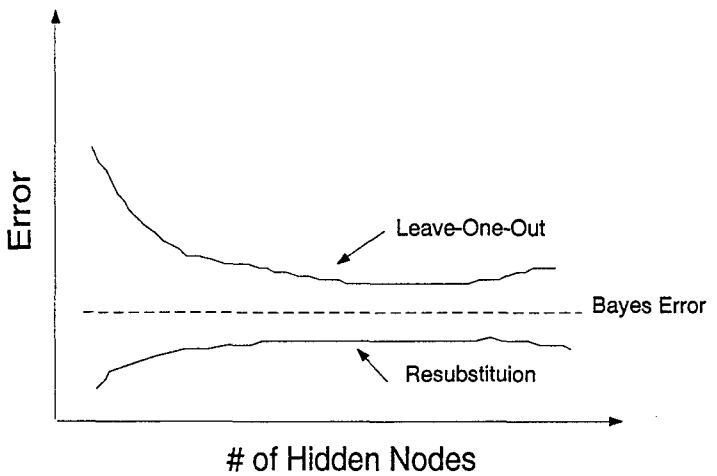



Figure 2.9 Bayes error bounding: Resubstitution and Leave-One-Out.

*2.6.2.1 Leave-One Out.* In the Leave-One-Out method, the entire data set minus one sample is used to train a classifier and the remaining sample is then used to test the classifier. This process is repeated until each sample has been left out and used to test the classifier. The error is proportion of the total number of test samples misclassified. Error estimates are performed in this manner over a range of a classifier parameter, such as number of hidden layer nodes. This produces an upper bound for Bayes error as shown in Figure 2.9.

*2.6.2.2 Resubstitution.* Using the Resubstitution method, the entire data set is used for both training and testing the classifier. The error is, again, the number of test samples misclassified and the error estimates are calculated for a range of classifier parameter values. The Resubstitution method produces a lower bound as shown in Figure 2.9.

## *2.7 Summary*

This chapter provides the theoretical basis for the methods used throughout this research effort. The use of neural networks for pattern recognition problems is discussed with particular attention given to the multi-layer perceptron. Furthermore, the importance of properly selecting the learning rate and the momentum constant for multi-layer perceptron classifiers and the need to reduce the input feature set in order to achieve a suitable classifier is stressed. Finally, the evaluation of such a classifier using Bayes error is also presented.

# *III. Methods & Results*

## *3.1 Introduction*

This chapter presents the methodology used to design and evaluate a multi-layer perceptron to classify the given environmental data sets. Because this methodology is sequential and some of the steps in the process require results from the previous step, the intermediate results and the final results are also presented in this chapter. All techniques discussed are based on the theory presented in Chapter II and are implemented in MATLAB®. Appendix B contains all MATLAB® functions employed in this research.

## *3.2 Data Description*

This thesis demonstrates the classifier design methodology using two data sets which were provided by AFTAC. The first data set consists of measurements of three classes of stainless steel: ss304, ss316, and ss400. The input features for the stainless steel data (shown in Table 3.1) are the percent by atom of six radioisotopes, percent by weight of the elements from Lithium to Californium, and the associated measurement errors.

| **Radioisotopes** | **% by Atom** | **Measurement Error** |
|---|---|---|
| Uranium 234 | $U_{234}P$ | $U_{234}E$ |
| Uranium 235 | $U_{235}P$ | $U_{235}E$ |
| Uranium 236 | $U_{236}P$ | $U_{236}E$ |
| Plutonium 240 | $Pu_{240}P$ | $Pu_{240}E$ |
| Plutonium 241 | $Pu_{241}P$ | $Pu_{241}E$ |
| Plutonium 242 | $Pu_{242}P$ | $Pu_{242}E$ |
| **Elements** | **% by Weight** | **Measurement Error** |
| Lithium thru Californium | $X$ | $XE$ |

Table 3.1 Stainless steel features. *Note: X is the elemental symbol for the elements between Lithium and Californium inclusive in the periodic table; a full listing of these elements and symbols is provided in Appendix C.*

This results in a 196-dimensional feature set. The features for the actinide data consist only of the radioactive particle percentages and the associated measurement errors (Table 3.2), resulting in an 18-dimensional feature set. Unlike the three-class stainless steel data, the actinide class membership may be determined in one of two ways. Due to the sensitive nature of AFTAC's mission, the samples are labeled by generic, hyphenated alphanumeric descriptors (*A-1, A-2, B-1, C-1, ..., M-1*) which represent the sample origin. Allowing each descriptor to represent a different class results in 55 classes. However, allowing the class structure to be based solely on the alphabetic part of the descriptors (*A, B, C, ..., M*) results in 15 classes. This research performs classification using both class structures. Therefore, essentially three data sets are used for classification, a stainless steel data set and two actinide data sets.

| **Radioisotopes** | **% by Atom** | **Measurement Error** | **% Measurement Error** |
|---|---|---|---|
| Uranium 234 | $U_{234}P$ | $U_{234}E$ | $U_{234}PE$ |
| Uranium 235 | $U_{235}P$ | $U_{235}E$ | $U_{235}PE$ |
| Uranium 236 | $U_{236}P$ | $U_{236}E$ | $U_{236}PE$ |
| Plutonium 240 | $Pu_{240}P$ | $Pu_{240}E$ | $Pu_{240}PE$ |
| Plutonium 241 | $Pu_{241}P$ | $Pu_{241}E$ | $Pu_{241}PE$ |
| Plutonium 242 | $Pu_{242}P$ | $Pu_{242}E$ | $Pu_{242}PE$ |

Table 3.2 Actinide features.

The data is configured in matrix form such that the first column contains an integer class designator, the remaining columns represent input features, and each row represents an individual sample. The configurations for each data set are given in Equations 3.1 - 3.3. In Equation 3.1, classes one, two, and three represent ss304, ss316, ss400, respectively. The full set of alphanumeric descriptors and the associated integer class values are given

in Appendix E.

$$Steel = \begin{bmatrix} \text{Class} & \text{Feature 1} & \text{Feature 2} & \cdots & \text{Feature 196} \\ 1 & U_{234}P & U_{235}P & \cdots & CfE \\ 1 & U_{234}P & U_{235}P & \cdots & CfE \\ \vdots & \vdots & \vdots & & \vdots \\ 3 & U_{234}P & U_{235}P & \cdots & CfE \\ 3 & U_{234}P & U_{235}P & \cdots & CfE \end{bmatrix} \tag{3.1}$$

$$Actinide_1 = \begin{bmatrix} \text{Class} & \text{Feature 1} & \text{Feature 2} & \cdots & \text{Feature 18} \\ 1 & U_{234}P & U_{234}E & \cdots & Pu_{242}PE \\ 1 & U_{234}P & U_{234}E & \cdots & Pu_{242}PE \\ \vdots & \vdots & \vdots & & \vdots \\ 55 & U_{234}P & U_{234}E & \cdots & Pu_{242}PE \\ 55 & U_{234}P & U_{234}E & \cdots & Pu_{242}PE \end{bmatrix} \tag{3.2}$$

$$Actinide_2 = \begin{bmatrix} \text{Class} & \text{Feature 1} & \text{Feature 2} & \cdots & \text{Feature 18} \\ 1 & U_{234}P & U_{234}E & \cdots & Pu_{242}PE \\ 1 & U_{234}P & U_{234}E & \cdots & Pu_{242}PE \\ \vdots & \vdots & \vdots & & \vdots \\ 15 & U_{234}P & U_{234}E & \cdots & Pu_{242}PE \\ 15 & U_{234}P & U_{234}E & \cdots & Pu_{242}PE \end{bmatrix} \tag{3.3}$$

### *3.3 Process Overview*

For each of the data sets described, the process illustrated in Figure 3.1 is used to develop a neural network classifier. The data is first preprocessed and the neural network architecture is selected. The parameters for the selected architecture and training algorithm are chosen. The network is trained, the feature set is reduced, and the classifier is trained again using this reduced set. Finally, the performance of the classifier is evaluated by comparing the accuracy of the classifier to the Bayes error bounds. The following sections provide a thorough treatment of each stage of the process.

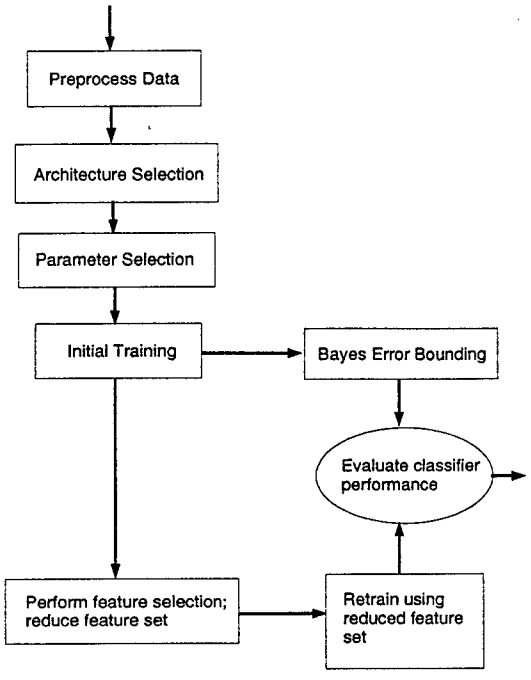



Figure 3.1 Process overview.

### *3.4 Data Preparation*

The data provided by AFTAC is preprocessed for two reasons. First, the data contains *gaps* where values for features in some samples were not given. Two methods of dealing with these data gaps are considered: (1) omit any samples which contain *gaps* or (2) replace missing values with zeros. Because there are many *gaps* in the data and because omitting entire samples means losing potentially useful data, the second option is employed. The data is also preprocessed to remove any features for which all values are identical; these homogeneous features are useless in classification (MATLAB® function *removeh.m*, Section B.1). This results in a stainless steel data set which contains only the 50 features shown in Table 3.3. Unlike the steel data, the actinide data set contains no homogeneous features and, therefore, remains unchanged.

After removing the homogeneous features, the remaining input features are normalized using the following simple normalization

$$data_{ij} = \frac{data_{ij} - \mu_j}{\sigma_j} \tag{3.4}$$

where $i$ represents the sample, $j$ represents the feature, $\mu_j$ is the mean of the feature $j$, and $\sigma_j$ is the standard deviation of feature $j$ (MATLAB® function *normal.m*, Section B.2). This is done to prevent classes with features of high magnitude from disproportionately affecting the weight update (Equation 2.9) during training.

| Features | | | |
|---|---|---|---|
| $U_{234}P$ | $U_{235}P$ | $U_{236}P$ | $U_{234}E$ |
| $U_{235}E$ | $U_{236}E$ | $Pu_{240}P$ | $Pu_{240}E$ |
| $Pu_{241}E$ | $Pu_{242}P$ | $C$ | $O$ |
| $F$ | $Na$ | $Mg$ | $Al$ |
| $Si$ | $P$ | $S$ | $Cl$ |
| $K$ | $Ca$ | $Ti$ | $V$ |
| $Cr$ | $Mn$ | $Fe$ | $Co$ |
| $Ni$ | $Cu$ | $Zn$ | $Zr$ |
| $Mo$ | $Ru$ | $Rh$ | $Ag$ |
| $Cd$ | $Sb$ | $Nd$ | $Sm$ |
| $Gd$ | $W$ | $Pb$ | $Bi$ |
| $Th$ | $U$ | $Pu$ | $OE$ |
| $CrE$ | $FeE$ | | |

Table 3.3 *Steel*: feature set after homogeneous feature removal.

### *3.5 Architecture Parameter Selection*

For reasons discussed in Section 2.4, the multi-layer perceptron is used for this research effort. Because the number of input features, $I$, and the number of output nodes,

$K$, are determined by the data set, the only free architecture parameters are the number of hidden layer nodes, $J$, and the type of activation functions to be used.

A larger number of hidden layer nodes allow decision regions of higher complexity to be drawn in the feature space but also diminishes the generalization capability of the multi-layer perceptron. This loss of generalization is due to the *memorization* of the training data allowed by a large number of hidden nodes. To insure good generalization, the number of training samples, $n$, in the data set should be at least ten times the number of weights in the network [2, 18]. Using this rule-of-thumb, the following upper bound for $J$ may be derived:

$$J < \frac{\frac{n}{10} - K}{I + K + 1} \tag{3.5}$$

Table 3.4 shows the suggested $J$ values using this rule-of-thumb along with the parameters actually used in implementing the classifiers. Because the suggested values for $J$ are low, a value of five is chosen arbitrarily for each data set with the intention to show, through demonstration, that good generalization and reasonable training times are possible using this number of hidden layer nodes.

| Data Set | $I$ | $K$ | $n$ | Suggested $J$ | Actual $J$ | Number of Weights |
|---|---|---|---|---|---|---|
| *StainlessSteel* | 50 | 3 | 473 | $< 1$ | 5 | 273 |
| $Actinide_1$ | 18 | 55 | 1151 | $< 1$ | 5 | 425 |
| $Actinide_2$ | 18 | 15 | 1151 | 2 | 5 | 185 |

Table 3.4 Multi-layer perceptron architecture parameters.

In addition to selecting the number of hidden layer nodes, the activation functions must be selected for both the hidden layer and the output layer. Although it has been suggested that the *Hyperbolic Tangent* has properties which may make it more desirable for training [14], the activation functions for all classifiers in this research are chosen to be *Sigmoid - Sigmoid.* That is, both the hidden layer nodes and the output layer nodes use the *Sigmoid* transformation function. AFIT has successfully implemented the *Sigmoid-Sigmoid* combination on other data sets [25].

## *3.6 Backpropagation Parameter Selection*

The backpropagation with momentum algorithm is used to train all multi-layer perceptrons in this research. The only parameters for which values must be chosen when using this method are the learning rate, $\eta$ , and the momentum, $\alpha$. Using standard backpropagation with momentum, learning rate and momentum remain constant throughout training. Because the quality of learning using this algorithm is highly dependent on these parameters, it is critical to properly select the learning rate and momentum values.

Fallside suggests training several classifiers using different combinations of learning rate and momentum values and selecting the parameter values which give rise to the smoothest learning curve (Classification Error versus Epoch) with the smallest error [7]. In general, a smoother learning curve indicates a more direct descent on the weight space error surface and it suggests more stable learning, which is less sensitive to the initial values of the weights. By plotting the learning curves of the different learning rate-momentum combinations on a single graph, the parameters which give rise to the best learning may be selected manually. These collections of learning curves are referred to collectively as Fallside plots.

For each of the data sets, learning rate and momentum are allowed to vary from .1 to .9 in .2 increments, producing twenty-five combinations of learning rate and momentum. Twenty-five classifiers, one for each combination of learning rate and momentum, are trained for each data set; all training is performed over 1000 epochs (MATLAB® function *fallside.m*, Section B.4). The data sets are not divided into training and test sets, rather the entire data sets are used for training each combination. One learning curve is produced for each combination of learning rate and momentum. In order to compare these twenty-five traces, five plots consisting of five traces are produced for each data set (MATLAB® function *postf.m*, Section B.5). For clarity, only the best and worst learning curves for each data set are shown here (Figures 3.2-3.4). The full set of Fallside plots for each data set are shown in Appendix D.

*3.6.1 Steel Results.* For the *Steel* data set (Figure 3.2), training with $\eta = \alpha = .1$ is clearly better than $\eta = .3$ and $\alpha = .9$. Although the latter curve contains very little

oscillation, it levels off at epoch 397 with a classification error of 8.2%. This classification error may indeed decline again if the network is trained for more epochs (200,000 epochs for example) but there is no reason to select these parameter values because the best case parameter values ($\eta = \alpha = .1$) produce a 0.63% error within the first 1000 epochs. The oscillations around epoch 200 and between epochs 600 and 800 are of little consequence with respect to the overall learning of the classifier using these parameters. Consequently, the best case parameters, $\eta = \alpha = .1$, are used for subsequent classifiers for the *Steel* data set.

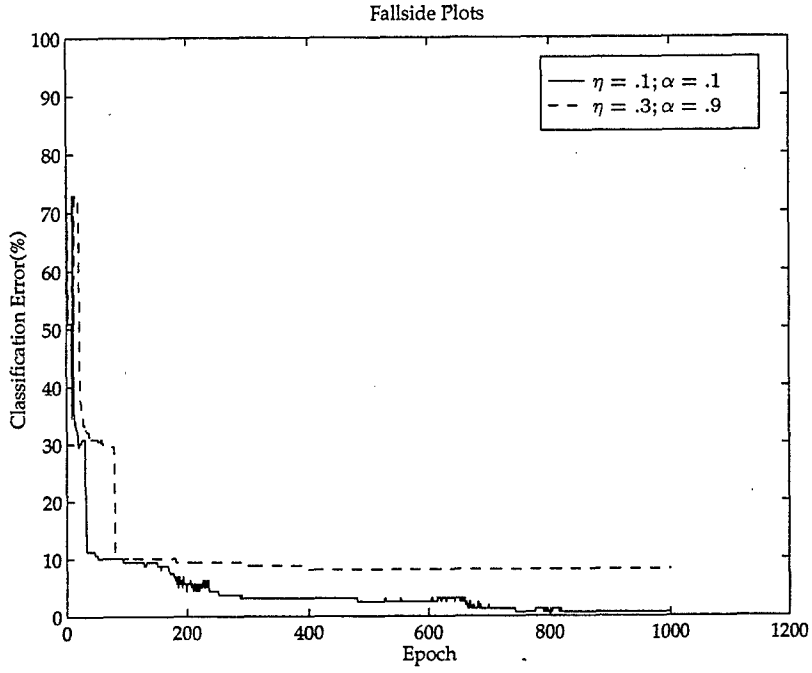



Figure 3.2 *Steel*: learning curves for the best and worst combinations of learning rate and momentum.

It should be noted that the parameter selection process may be repeated using a refined range of values such as (0,.1). The accuracy will be no better, on average, if the error rates are already within the Bayes error bounds. It may, however, provide some improvement in training times.

*3.6.2 Actinide$_1$ Results (55-class).* Unlike the best *Steel* learning curve which approaches .60% classification error, none of the *Actinide*$_1$ classifiers (Figure 3.3) perform better than 60% error. The worst learning curve ($\eta = .9; \alpha = .3$) reaches a minimum of 86% error under 200 epochs, while the best learning ($\eta = .1; \alpha = .5$) curve reaches 60% error by epoch 800 at which point it levels off and begins oscillating. Although this level of error is not acceptable for a final classifier, the goal here is to select the best combination of parameters to continue in the design of the classifier. The overall performance of the classifier is evaluated in later analysis. Therefore, a learning rate of .1 and a momentum of .5 are selected for subsequent classifiers for this data set.

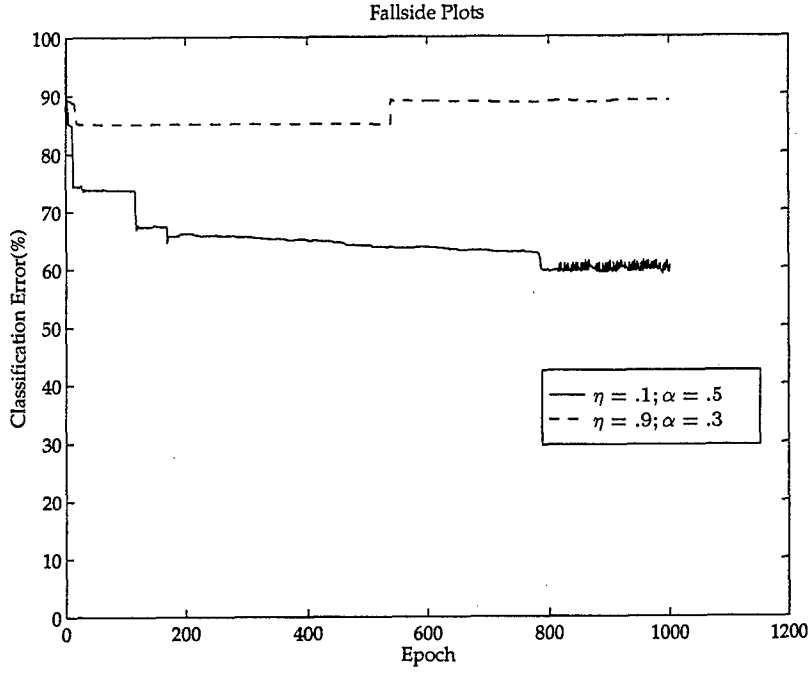



Figure 3.3 *Actinide*$_1$ (55-class): learning curves for the best and worst combinations of learning rate and momentum.

*3.6.3 Actinide$_2$ Results (15-class).* With the 15-class *Actinide*$_2$ data set, the error rates achieved by all of the classifiers are lower than those of the *Actinide*$_1$ classifiers. The best learning curve ($\eta = \alpha = .1$) approaches 20% error but oscillates heavily throughout training. Though this is not usually desirable, all of the Fallside traces which give comparable error values also oscillate heavily (see Appendix D). The poorest learning

occurs with $\eta = \alpha = .9$ in which the error drops immediately to 45% and where it remains throughout the remainder of the training. Based on these plots, the learning rate and the momentum are both set to .1 for further classifier training.

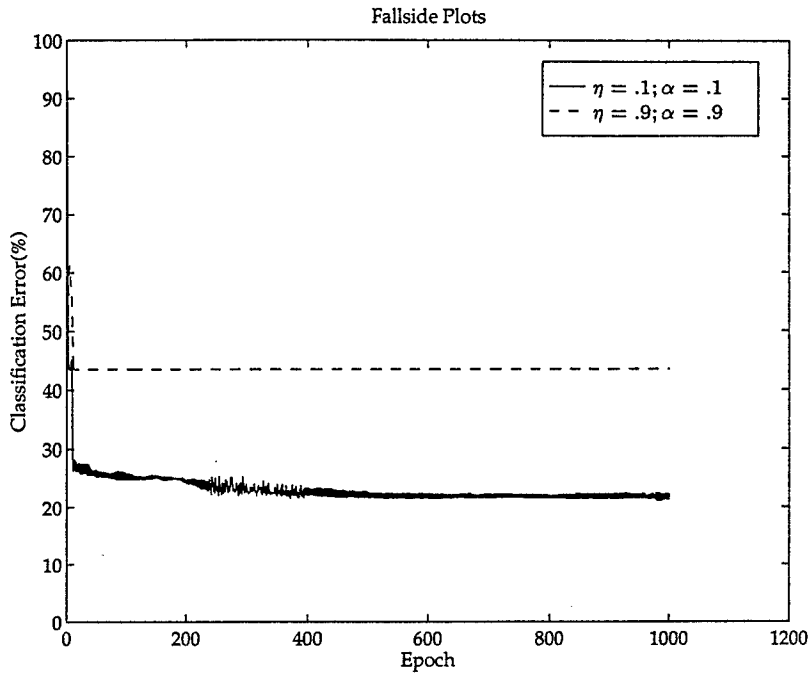



Figure 3.4 *Actinide*$_2$ (15-class): learning curves for three combinations of learning rate and momentum.

### *3.7 Initial Training*

Once the parameters for each data set are selected, initial training is performed. This training provides an initial indication of the performance of the classifiers, and allows the selection of appropriate training times. By training a multi-layer perceptron for a large number of epochs, the training data may be *memorized.* That is, the network can classify the entire training set with 100% accuracy. Initially this may seem to be a desirable outcome, however, when this is done the network loses its ability to generalize (classify samples which were not used in training). The number of training epochs should be selected at the point where this loss of generalization begins to occur.

The method used to determine the point at which the network begins to lose its generalization ability is to divide the data set into two sets (training and test) and track the classification error for the test set while training the network with the training set. After each epoch of training, each sample in the test set is classified using the current set of weights and the percentage of samples misclassified is stored (MATLAB® function *tev.m*, Section B.8). The number of training epochs is chosen where the test set error begins to increase.

As noted above, the data sets are each divided into two subsets, training and testing (MATLAB® function *randchoose.m*, Section B.3). The training subset of each data set contains two thirds of the samples by class, while the test set contains the remaining third by class. This splitting of the data sets for training and testing presents a slight problem in the case of the actinide data sets. In these data sets, many of the classes contain only one sample. For the classes in which this is true, the sample is duplicated in both the training and test subsets.

In order to insure that the results obtained by this method are insensitive to the initial values of the weight matrices, each data set is trained ten times beginning each training run with different initial weight matrices (randomly chosen values as discussed in Section 2.4.2). The *Steel* classifier is trained for 500 epochs on each run, while the $Actinide_1$ and the $Actinide_2$ classifiers are trained for 20,000 epochs on each run. The mean of the training curves and the mean of the test curves for each data set are shown in Figures 3.5-3.7.

*3.7.1 Steel Results.* For the *Steel* data, the test set error reaches a minimum of 11% at epoch 65 at which point it begins to increase and the training set error is still declining. Therefore, the number of epochs for training the *Steel* data is selected as 65. It should be noted that this value is the point at which the *average* curve is at a minimum. The individual training curves reach minimum error at different points in training. The range of epochs at which the minimum error is reached for the different learning curves is [25 180].

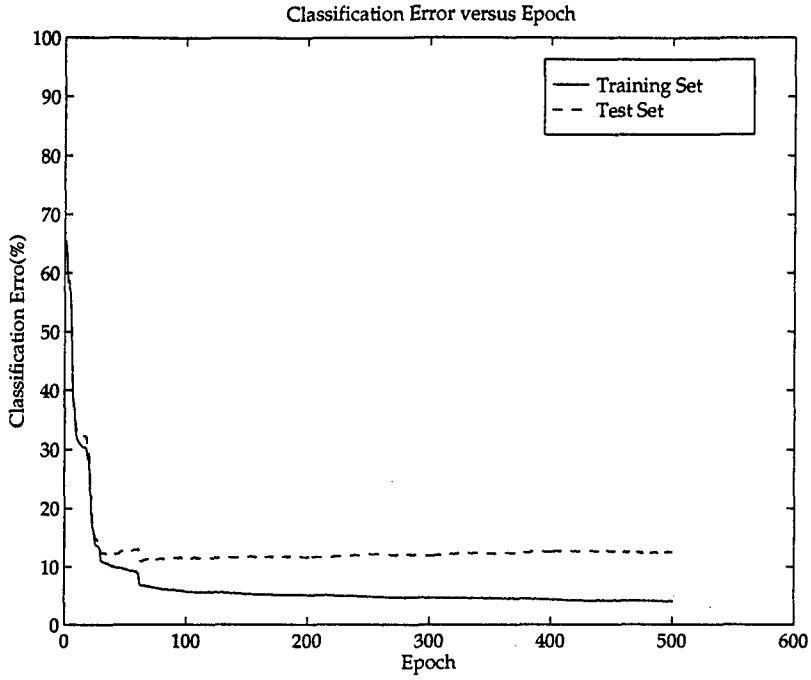



Figure 3.5 *Steel*: training and test set error traces over 500 epochs.

The confusion matrices after 65 training epochs for both the training and the test sets are computed (Tables 3.5 and 3.6). Each row in these matrices indicates the actual class of samples in the data set, while columns represent the class assigned to the samples by the classifier. A value in the table at row $i$ and column $j$ indicates the number of samples of class $i$ which are classified as class $j$ by the classifier. Therefore, values on the main diagonal represent the number of samples correctly classified in each class. For example, in Table 3.5, 148 samples of class one are classified as class one, while six are misclassified as class two and two are misclassified as class three.

| | **Assigned Class** | | | |
|---|---|---|---|---|
| **Actual Class** | 1 | 2 | 3 | **% Correct** |
| 1 | 148 | 6 | 2 | 95 |
| 2 | 4 | 78 | 0 | 95 |
| 3 | 0 | 0 | 76 | 100 |

Table 3.5 *Steel*: confusion matrix for the training set.

| Actual Class | Assigned Class | | | % Correct |
|---|---|---|---|---|
| | 1 | 2 | 3 | |
| 1 | 69 | 9 | 0 | 88 |
| 2 | 9 | 33 | 0 | 79 |
| 3 | 1 | 2 | 36 | 92 |

Table 3.6 *Steel*: confusion matrix for the test set.

The purpose of the confusion matrices is to illustrate the accuracy by class. For a data set in which the samples are not evenly distributed among the classes, the classifier may sacrifice accuracy in the classes which contain few samples, while achieving high accuracy in classes which contain many samples and maximizing the overall accuracy. The *Steel* samples are evenly distributed over the classes and the accuracy is consistent across the classes.

*3.7.2 Actinide$_1$ Results (55-class).* The *Actinide$_1$* training and test set error curves (Figure 3.6) asymptotically approach 57% and 59% respectively. At 20,000 epochs, the error curves are still on a downward trend but have leveled off significantly by this point. As a result, these error rates are considered to be indicative of the classifier performance on this data set.

The poor performance of the multi-layer perceptron classifier on this data set is due to the large number of classes and the small number of samples per class. Although the confusion matrices are too large to display (55x55), Tables 3.7 and 3.8 show the number of samples per class and the percentage of samples misclassified within each class. It is obvious that the more heavily populated classes are classified more accurately and that, in general, classes containing less than ten samples have zero accuracy. According to Foley, if the ratio of the number of samples per class to the number of features is greater than three, the training set error is approximately the test set error and they both approach Bayes error rate [10]. For the *Actinide$_1$* data set, there are only 18 features and the average number of samples per class is 21 with only six of the 55 classes containing more than 54 samples (the number of samples per class required to meet the Foley ratio requirement for the given

data). Therefore, it is doubtful that further training of the network will improve either the training set error or the test set error. As a result, further analysis of this data, with the exception Bayes error bounding, is abandoned in favor of the 15-class $Actinide_2$ data. Bayes error bounding is still performed on the $Actinide_1$ data set (Section 3.9) to confirm the assertion that additional training will not significantly increase the performance of the classifier using this data.

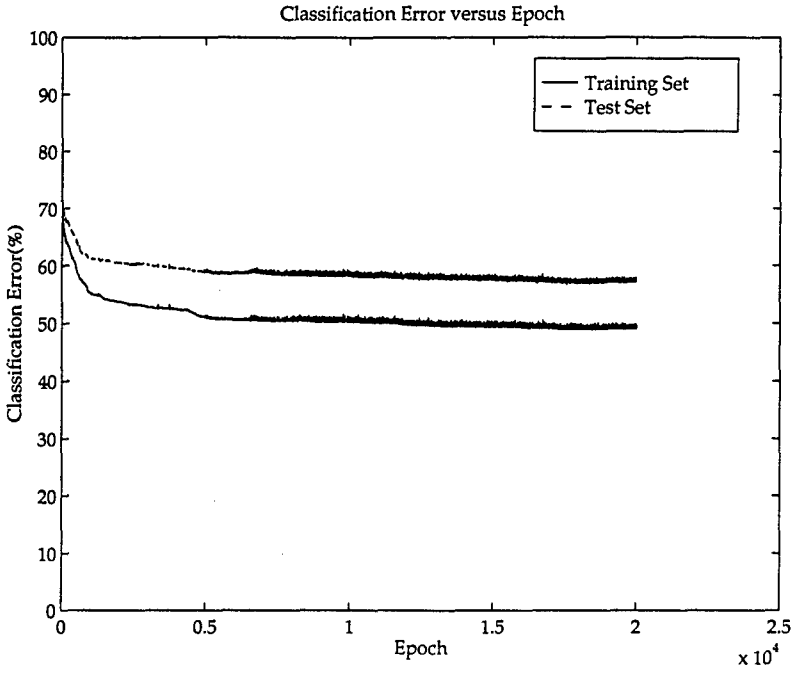



Figure 3.6 $Actinide_1$ (55-class): training and test set error traces plotted over 20,000 epochs.

| Class | Samples/Class | % Correct | Class | Samples/Class | % Correct |
|---|---|---|---|---|---|
| 1 | 6 | 0 | 28 | 2 | 100 |
| 2 | 1 | 0 | 29 | 12 | 0 |
| 3 | 10 | 0 | 30 | 12 | 17 |
| 4 | 1 | 0 | 31 | 1 | 0 |
| 5 | 1 | 0 | 32 | 1 | 0 |
| 6 | 1 | 0 | 33 | 2 | 0 |
| 7 | 2 | 0 | 34 | 4 | 0 |
| 8 | 1 | 0 | 35 | 2 | 0 |
| 9 | 1 | 0 | 36 | 1 | 0 |
| 10 | 4 | 0 | 37 | 1 | 0 |
| 11 | 4 | 0 | 38 | 1 | 0 |
| 12 | 48 | 90 | 39 | 1 | 0 |
| 13 | 1 | 0 | 40 | 1 | 0 |
| 14 | 6 | 0 | 41 | 1 | 0 |
| 15 | 2 | 0 | 42 | 14 | 0 |
| 16 | 1 | 0 | 43 | 1 | 0 |
| 17 | 4 | 0 | 44 | 2 | 0 |
| 18 | 1 | 0 | 45 | 6 | 0 |
| 19 | 1 | 0 | 46 | 16 | 69 |
| 20 | 1 | 0 | 47 | 1 | 0 |
| 21 | 2 | 0 | 48 | 8 | 0 |
| 22 | 50 | 4 | 49 | 144 | 94 |
| 23 | 58 | 79 | 50 | 1 | 0 |
| 24 | 16 | 0 | 51 | 8 | 0 |
| 25 | 96 | 68 | 52 | 1 | 0 |
| 26 | 48 | 33 | 53 | 1 | 0 |
| 27 | 130 | 58 | 54 | 8 | 0 |
| | | | 55 | 1 | 0 |

Table 3.7 *Actinide*$_1$ (55-class): training set accuracy by class.

| Class | Samples/Class | % Correct | Class | Samples/Class | % Correct |
|---|---|---|---|---|---|
| 1 | 5 | 0 | 28 | 2 | 0 |
| 2 | 1 | 0 | 29 | 6 | 0 |
| 3 | 5 | 0 | 30 | 7 | 0 |
| 4 | 1 | 0 | 31 | 1 | 0 |
| 5 | 1 | 0 | 32 | 1 | 0 |
| 6 | 1 | 0 | 33 | 2 | 0 |
| 7 | 1 | 0 | 34 | 4 | 0 |
| 8 | 1 | 0 | 35 | 1 | 0 |
| 9 | 1 | 0 | 36 | 1 | 0 |
| 10 | 3 | 0 | 37 | 1 | 0 |
| 11 | 2 | 0 | 38 | 1 | 0 |
| 12 | 26 | 85 | 39 | 1 | 0 |
| 13 | 1 | 0 | 40 | 1 | 0 |
| 14 | 3 | 0 | 41 | 1 | 0 |
| 15 | 3 | 0 | 42 | 7 | 0 |
| 16 | 1 | 0 | 43 | 1 | 0 |
| 17 | 3 | 0 | 44 | 3 | 0 |
| 18 | 1 | 0 | 45 | 3 | 0 |
| 19 | 1 | 0 | 46 | 10 | 60 |
| 20 | 1 | 0 | 47 | 1 | 0 |
| 21 | 1 | 0 | 48 | 6 | 0 |
| 22 | 27 | 0 | 49 | 74 | 86 |
| 23 | 29 | 76 | 50 | 1 | 0 |
| 24 | 8 | 0 | 51 | 5 | 0 |
| 25 | 48 | 54 | 52 | 1 | 0 |
| 26 | 26 | 8 | 53 | 1 | 0 |
| 27 | 65 | 49 | 54 | 4 | 0 |
| | | | 55 | 1 | 0 |

Table 3.8 *Actinide*$_1$ (55-class): test set accuracy by class.

*3.7.3 Actinide$_2$ Results (15-class).* Unlike the *Actinide*$_1$ learning curves, the training and test error traces (Figure 3.7) for the *Actinide*$_2$ data drop to 21% and 26%, respectively, within the first 3,000 epochs, but each only drops an additional .1% over the next 17,000 epochs. Although the minimum error in the test error curve (24%) occurs at epoch 15,399, training for this number of epochs greatly increases the computation time on a data set of this size (1151 samples) without a significant decrease in the test set error. To avoid this increase in computation time, the number of training epochs is selected as 3,000.

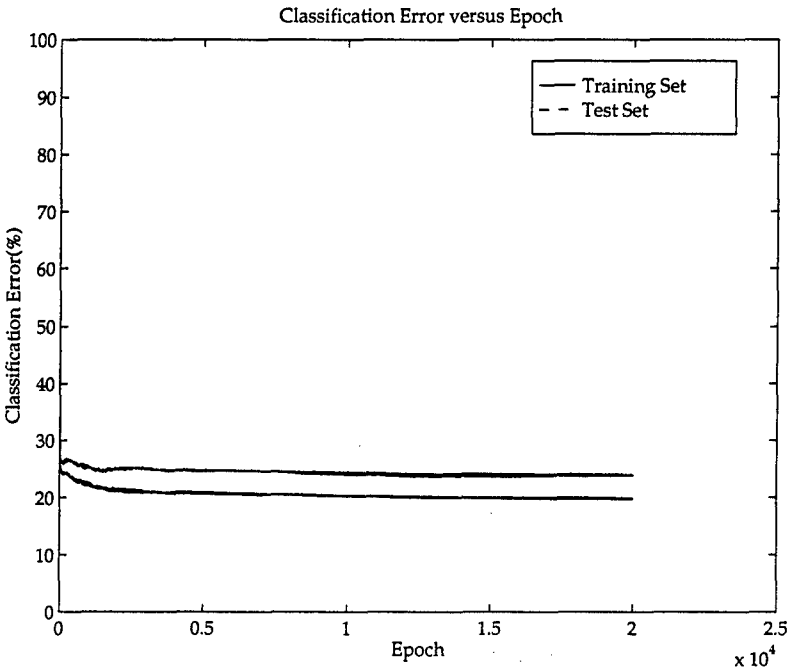



Figure 3.7 *Actinide*$_2$ (15-class): training and test set error traces over 20,000 epochs.

The confusion matrices for the *Actinide*$_2$ data set are shown in Tables 3.9 and 3.10. The accuracy is clearly lower in the classes which contain fewer samples. Classes 12 and 13, which contain the majority of the samples in the entire data set, dominate the training and, as a result, the classifier performs well on these classes in both the training and test sets. In addition, the next most populated class, class 5, is the only other class with an accuracy greater than zero. This suggests that training may be more successful on the remaining classes if they are more heavily populated.

| Actual Class | Assigned Class | | | | | | | | | | | | | | | % Correct |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | |
| 1 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 17 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 22 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 23 | 0 | 0 | 46 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 6 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 197 | 23 | 0 | 0 | 90 |
| 13 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 44 | 385 | 0 | 0 | 89 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

Table 3.9 *Actinide*$_2$ (15-class): confusion matrix for the training set.

| Actual Class | Assigned Class | | | | | | | | | | | | | | | % Correct |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 11 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 17 | 0 | 0 | 35 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 94 | 18 | 0 | 0 | 87 |
| 13 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 28 | 183 | 0 | 0 | 84 |
| 14 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 0 | 0 | 0 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

Table 3.10 *Actinide*$_2$ (15-class): confusion matrix for the test set.

*3.7.4 Additional Analysis.* It should be noted that two other neural network algorithms, fuzzyARTmap [5] and Multi-layer perceptron Iterative Construction Algorithm (MICA, developed at AFIT)[22], were used to classify the actinide data sets. The fuzzyARTmap architecture, which is particularly well suited for data with only a few samples per class, performs no better than the multi-layer perceptron on either the 55-class *Actinide*$_1$ data or the 15-class *Actinide*$_2$. Conversely, the MICA network, which adds hidden layer nodes until it is able to separate the training data, classifies the *Actinide*$_2$ training set data with 100% accuracy but it requires 851 hidden layer nodes to do so and the test error is 92%. This indicates that the feature set for this data is not sufficient to separate the data and maintain generalization; good features are required to produce good classifiers. Although the 55-class *Actinide*$_1$ classification using MICA was not completed

due to computation time, the error rates would be higher because of the larger number of classes. From this additional analysis, it is concluded that the poor performance of the multi-layer perceptron on the data is due not only to the large number of classes and the small number of samples per class but also to overlap in the feature space.

## *3.8 Feature Reduction & Classifier Retraining*

Once initial training is complete, the features for each data set are ranked according to their saliency, the least salient features in each data set are removed, and the classifiers are retrained using the reduced data sets. As discussed in Section 3.7.2, the $Actinide_1$ data set is not considered in this section.

The general process in this portion of the research is to rank order the features by forward sequential selection and plot the error associated with the addition of each feature to the feature nucleus (MATLAB® function *fselct.m*, Section B.9). As each additional feature is added to the nucleus, the classification error decreases. At some point, a minimum error is reached. The number of features, $f$, at this point is the number of features which should be used to classify the given data set. Given this, the first $f$ features of the nucleus are considered to be the salient features and all other features are removed from the data set under consideration. Finally, a classifier, using the parameters determined previously, is trained on the reduced data set.

*3.8.1 Steel Results.* The classifiers in the forward sequential search are trained for 65 epochs with a learning rate and a momentum of .1. The resulting ranking of features is shown in Table 3.11. Figure 3.8 shows the classification error as features are added to the nucleus. Labels on the x-axis correlate to the ranks in Table 3.11. For example, after the first pass through all of the features, the addition of $Ni$ to the nucleus (which is initially empty) results in the best classification error (24.1%). So, $Ni$ is added to the nucleus. On the second pass, the addition of $Mo$ to the nucleus (which now contains $Ni$) results in the best classification error (6.8%). The minimum error (3.8%) is achieved with the addition of feature 34 to the nucleus. Therefore, the feature set is reduced to include only the top 34 features out of the original 50 features.

After selecting an appropriate feature set, the classifier is retrained on the reduced feature set. Although the classifier could be satisfactorily trained in 65 epochs, it is trained for 500 epochs in order to compare the results to Figure 3.5. Making this comparison, it is clear that reducing the feature set does not diminish the performance of the classifier. On the contrary, reducing the feature set improves the training and test set error (at epoch 500) from 4% to .6% and 13% to 5.5%, respectively.

| Rank | Symbol | Rank | Symbol |
|---|---|---|---|
| 1 | $Ni$ | 26 | $K$ |
| 2 | $Mo$ | 27 | $Co$ |
| 3 | $U_{234}P$ | 28 | $Cu$ |
| 4 | $Mg$ | 29 | $Ti$ |
| 5 | $Cr$ | 30 | $Mn$ |
| 6 | $Pu_{240}P$ | 31 | $Fe$ |
| 7 | $U_{236}E$ | 32 | $Ag$ |
| 8 | $S$ | 33 | $Zn$ |
| 9 | $U_{235}P$ | 34 | $Sb$ |
| 10 | $Ca$ | 35 | $Ru$ |
| 11 | $Zr$ | 36 | $Nd$ |
| 12 | $U_{236}P$ | 37 | $V$ |
| 13 | $U_{234}E$ | 38 | $Pb$ |
| 14 | $Pu_{241}E$ | 39 | $Cd$ |
| 15 | $U_{235}E$ | 40 | $Sm$ |
| 16 | $Pu_{240}E$ | 41 | $Th$ |
| 17 | $Na$ | 42 | $U$ |
| 18 | $F$ | 43 | $Pu$ |
| 19 | $Si$ | 44 | $P$ |
| 20 | $Pu_{242}P$ | 45 | $Gd$ |
| 21 | $Al$ | 46 | $W$ |
| 22 | $C$ | 47 | $Bi$ |
| 23 | $Cl$ | 48 | $OE$ |
| 24 | $O$ | 49 | $CrE$ |
| 25 | $Rh$ | 50 | $FeE$ |

Table 3.11 *Steel*: features ranked by saliency.

Apparent abnormalities in the feature ranking, such as the error in a measurement ranking higher than the measurement itself, may be resolved by running feature selection

multiple times and determining the relative frequency of each feature's ranks. Doing this, Ruck *et al* showed, for a given data set, that the ranking of the most salient and the least salient features were more consistent than features in the middle of the rankings [27]. The rankings of features in the middle varied widely over feature selection runs. For example, the top five and the bottom five *Steel* features may rank consistently in the top and bottom, respectively, over multiple runs. However, the rank of the middle features (feature 6 to feature 45 in Table 3.11) may vary widely over multiple runs. Therefore, it may be insignificant that $U_{236}E$ outranks $U_{236}P$ for the one feature selection run shown. Multiple runs could be used to verify the rankings.

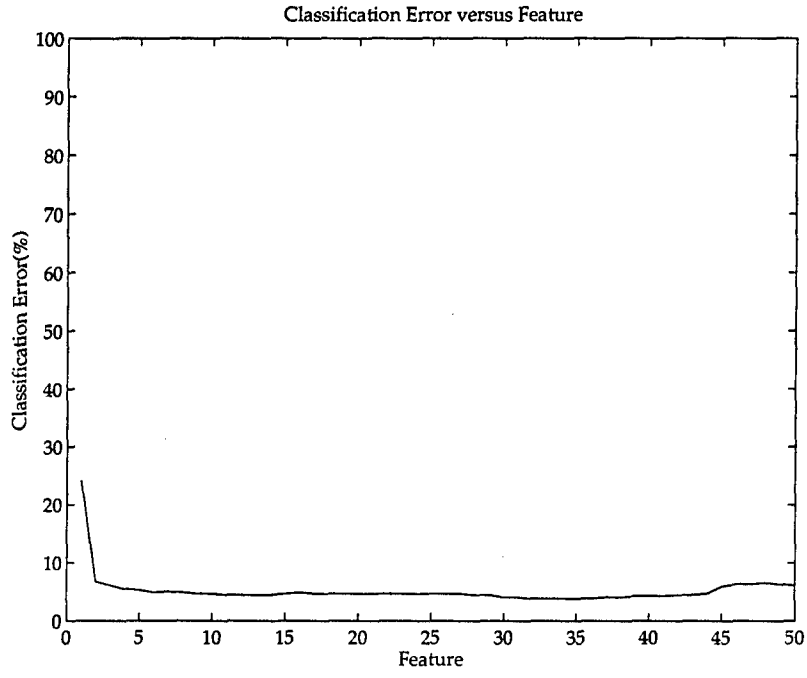



Figure 3.8 *Steel*: classification error versus feature added to the nucleus during forward sequential selection.

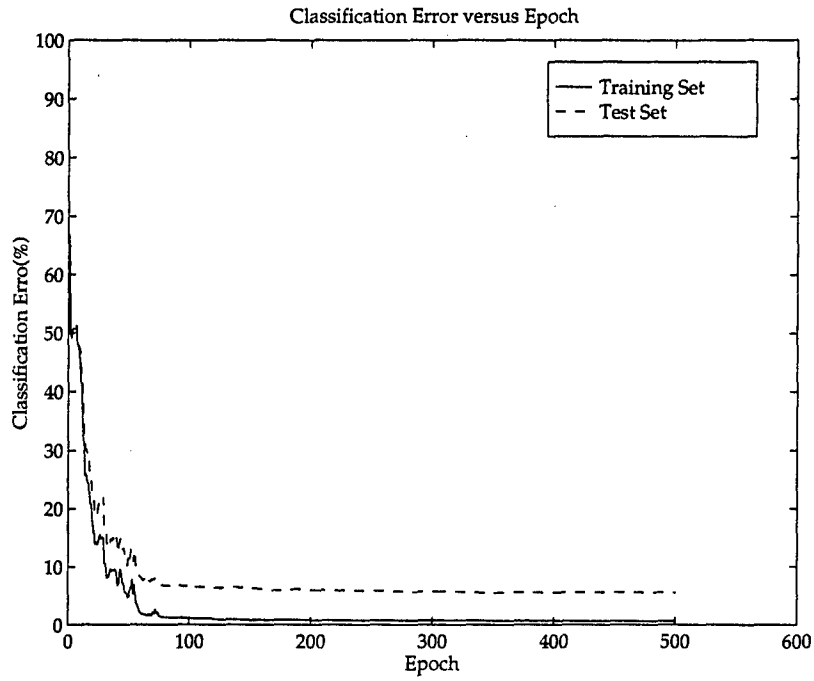



Figure 3.9 *Steel*: training and test set error traces over 500 epochs.

The confusion matrices are shown for one training run on the reduced feature set (Tables 3.12 and 3.13). Using the reduced feature set, the classifier shows marked improvement on all classes of both the training and the test sets.

| | Assigned Class | | | |
|---|---|---|---|---|
| **Actual Class** | 1 | 2 | 3 | **% Correct** |
| 1 | 155 | 1 | 0 | 99 |
| 2 | 0 | 82 | 0 | 100 |
| 3 | 0 | 0 | 76 | 100 |

Table 3.12 *Steel* (reduced feature set): confusion matrix for the training set.

| Actual Class | Assigned Class | | | % Correct |
|---|---|---|---|---|
| | 1 | 2 | 3 | |
| 1 | 71 | 2 | 5 | 91 |
| 2 | 2 | 40 | 0 | 95 |
| 3 | 1 | 0 | 38 | 97 |

Table 3.13 *Steel*(reduced feature set): confusion matrix for the test set.

Overall the feature selection is successful. The feature set is reduced from 50 features to 34 features with no decrease in the overall accuracy of the classifier. In fact, on this data set, the reduced feature set provides better classification than the full feature set. This improvement in accuracy results from omission of features which, in terms of classification, are analogous to noise. As with most systems, noise results in diminished performance.

*3.8.2 Actinide$_2$ Results (15-class).* The classifiers in the forward sequential selection on the *Actinide$_2$* data are trained for 3,000 epochs. Table 3.14 shows the features ranked by saliency. Again, plotting the features against the classification error, as shown in Figure 3.10, indicates an appropriate number of features. In this case, the ninth feature added to the nucleus results in the lowest overall classification error. Therefore, the feature set is reduced to the first nine features listed in Table 3.14.

The classifier is retrained on the reduced feature set. The training and test set learning curves are shown in Figure 3.11. The overall error of both the training set (24% error) and test set (26%) is higher than the associated error when using the full feature set. However, the ultimate goal of feature selection/reduction is to reduce the feature set to reduce training times without significantly reducing the accuracy of the classifier. If the reduced feature set error is greater than that of the full feature set error, but both are less than the upper bound on Bayes error, the difference in the accuracy using the two feature sets may be considered insignificant. This comparison is made in Section 3.9 where the Bayes error bounding procedure and results are presented.

| Rank | Symbol | Rank | Symbol |
|---|---|---|---|
| 1 | $U_{236}P$ | 10 | $Pu_{240}E$ |
| 2 | $U_{234}PE$ | 11 | $U_{235}E$ |
| 3 | $Pu_{242}P$ | 12 | $Pu_{241}E$ |
| 4 | $Pu_{240}PE$ | 13 | $U_{234}P$ |
| 5 | $Pu_{240}P$ | 14 | $U_{236}PE$ |
| 6 | $U_{236}E$ | 15 | $U_{235}P$ |
| 7 | $Pu_{242}E$ | 16 | $Pu_{241}PE$ |
| 8 | $Pu_{242}PE$ | 17 | $U_{235}PE$ |
| 9 | $Pu_{241}P$ | 18 | $U_{234}E$ |

Table 3.14 *Actinide*$_2$ (15-class): features ranked by saliency.

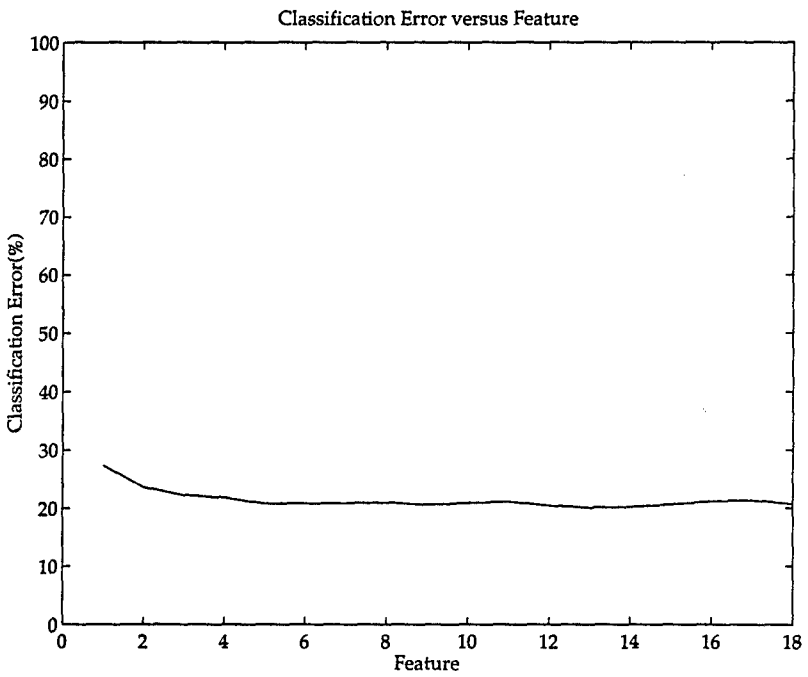



Figure 3.10 *Actinide*$_2$ (15-class): Classification error versus feature added to the nucleus during forward sequential selection.

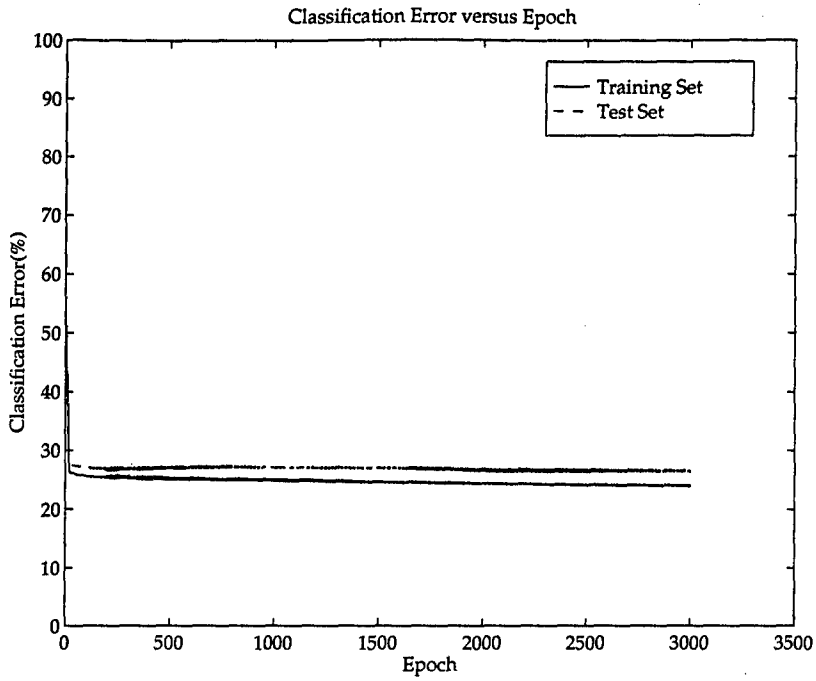



Figure 3.11 *Actinide*$_2$ (15-class): training and test set error traces over 3,000 epochs.

The confusion matrices for the *Actinide*$_2$ data set are shown in Tables 3.15 and 3.16. The confusion matrices show that there are shifts in the by-class accuracies in both the training and test sets. However, the shift in overall accuracy is less than 1% in both sets. Although this seems to be contrary to the decrease in the error of both sets as demonstrated by Figures 3.7 and 3.11, it should be noted that the confusion matrices are computed for one training run, while each curve in the figures represent an average of ten classifier training runs. The confusion matrices show that, even with the reduced feature set, the classifier still focuses on and performs well on the classes which are more heavily populated.

| Actual Class | Assigned Class | | | | | | | | | | | | | | | % Correct |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 20 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 40 | 0 | 0 | 12 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 4 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 6 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 189 | 31 | 0 | 0 | 98 |
| 13 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 44 | 385 | 0 | 0 | 89 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 4 | 2 | 0 | 25 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

Table 3.15 *Actinide*$_2$ (reduced feature set): confusion matrix for the training set.

| Actual Class | Assigned Class | | | | | | | | | | | | | | | % Correct |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 11 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 24 | 0 | 0 | 8 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 97 | 15 | 0 | 0 | 87 |
| 13 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 23 | 191 | 1 | 0 | 88 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 0 | 0 | 0 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

Table 3.16 *Actinide*$_2$ (reduced feature set): confusion matrix for the test set.

### *3.9 Bayes Error Bounding & Classifier Evaluation*

In order to evaluate the performance of the classifiers, the error rates achieved are compared to the Bayes error bounds for each data set. Because Bayes error rate represents the best performance that any classifier can achieve on average, the classifier is considered to be a *good* classifier for the given data set if its test set error falls within the Bayes error bounds.

In estimating the Bayes error bounds for the given data sets, only the upper bound (determined by the Leave-One-Out Method) is calculated. The lower bound using the multi-layer perceptron is rather useless because it approaches zero as the number of hidden nodes and the training times are increased[19]. Because Bayes error bounding is so

computationally intense, the upper bound is computed using one to ten hidden layer nodes for each of the data sets.

*3.9.1 Steel.* The Bayes error bound for the *Steel* data is shown in Figure 3.12. Over the range of hidden layer nodes from 3 to 6, the error remains at approximately 10%. Both the training set error (.60%) and the test set error (5.5%) are clearly less than the 10% upper bound. Therefore, the multi-layer perceptron performs as well, on average, as any other classifier, statistical or neural, for the *Steel* data set.

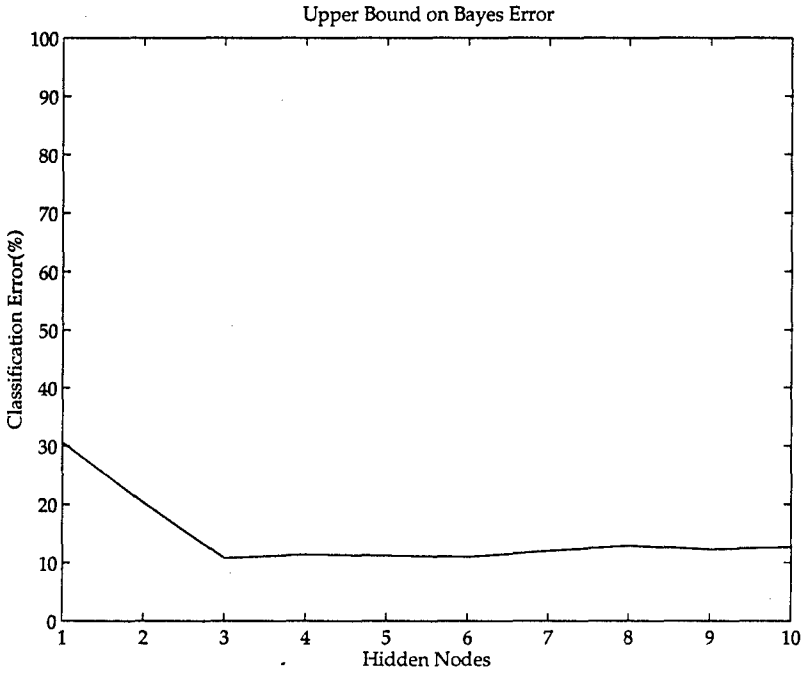



Figure 3.12 *Steel*: Bayes error bounds.

*3.9.2 Actinide$_1$ (55-class).* The upper bound on Bayes error for the *Actinide$_1$* data set is shown in Figure 3.13. At five hidden layer nodes, the upper bound on Bayes error is 63%. Comparing this to the training error (57%) and the test error (59%), the classifier performance approaches Bayes optimality for the data set as given. This confirms the assertion made in Section 3.7.2 that further training will not significantly increase the accuracy on this 55-class data structure. Therefore, the decision to drop the *Actinide$_1$* data set from consideration is fully justified.

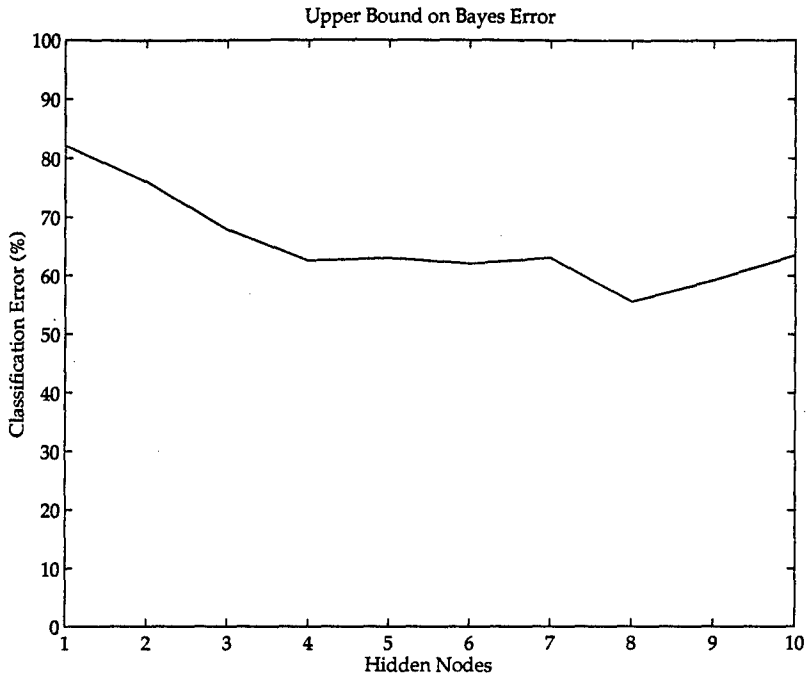



Figure 3.13 *Actinide*$_1$ (55-class): Bayes error bounds.

*3.9.3 Actinide*$_2$ *(15-class).* The Bayes error bound for the *Actinide*$_2$ data set is shown in Figure 3.14. At five hidden layer nodes, the upper bound on Bayes error is 29%. Using the full feature set, the training and test set errors (21% and 26%, respectively) are within this upper bound. Furthermore, the reduced feature set classification error also approaches Bayes error, though it results in slightly higher training and test set errors (24% and 26%, respectively). Because Bayes error represents the best that any classifier can perform on average on a given data set, the features measured and the number of samples per class in the *Actinide*$_2$ data are sufficient to allow only 74% to 79% accuracy.

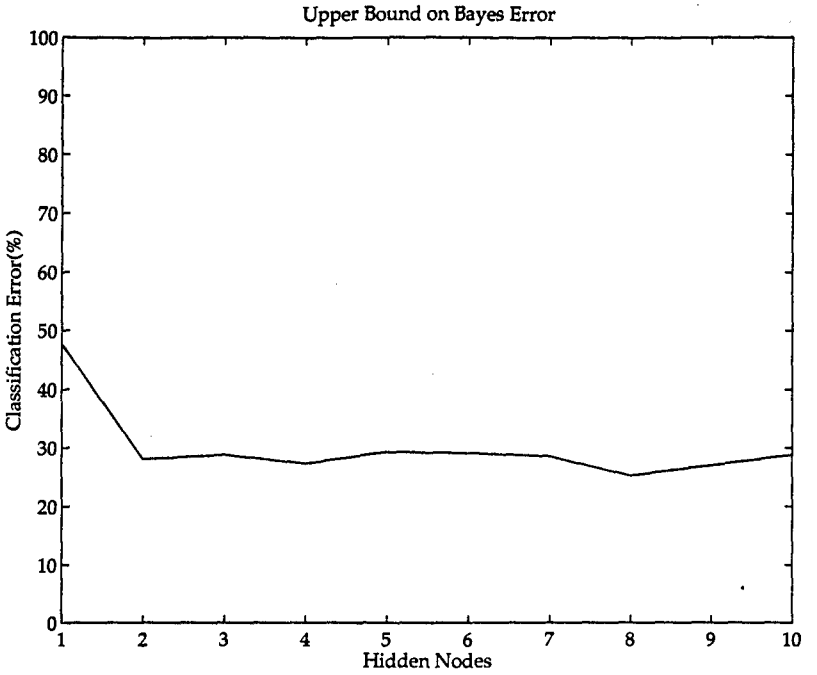



Figure 3.14 *Actinide*$_2$ (15-class): Bayes error bounds.

## *3.10 Summary*

This chapter has presented the methods used to design multi-layer perceptron classifiers for three data sets as well as all results obtained by these methods. A classifier was designed for each data set by selecting the appropriate classifier parameters, conducting initial training, reducing the feature set/retraining, and comparing the performance of each to Bayes error. Table 3.17 shows the final results of this research effort.

| Data Set | # Features | Training Error | Test Error | Bayes Error |
|---|---|---|---|---|
| *Steel* | 50 | .6% | 5.5% | 11% |
| *Actinide*$_1$ | 18 | 57% | 59% | 63% |
| *Actinide*$_2$ | 9 | 24% | 26% | 29% |

Table 3.17 Summary of results: the Bayes error value is the upper bound at five hidden layer nodes.

The training and test set errors are the lowest average values of ten training runs with each run beginning with different random weight matrices. The Bayes error values are the upper bound estimates using five hidden layer nodes. The results are fully summarized and conclusions are drawn in Chapter IV.

# *IV. Conclusion*

## *4.1 Introduction*

The primary objective of this research is to provide a methodology by which environmental professionals may design neural networks to classify environmental samples. The secondary goals are to design a multi-layer perceptron to classify two environmental data sets which were provided by AFTAC, to determine the salient features for the given data sets, and to evaluate the performance of each classifier produced.

## *4.2 Methodology Summary*

The methodology implemented in this research may be summarized as follows:

1. Select the architecture parameters.
2. Select the training parameters for the given data using Fallside plots.
3. Train the classifier tracking the test set error and select an appropriate number of training epochs.
4. Perform forward sequential selection and reduce the feature set.
5. Train the classifier again using the reduced feature set.
6. Calculate the Bayes error bounds and evaluate the classifier performance.

## *4.3 Summary of Results*

*4.3.1 Stainless Steel.* The stainless steel data provided by AFTAC consists of 196 input features. By removing the homogeneous features, the data is pared down to 50 features. Using Fallside plots, the learning rate and momentum for the backpropagation algorithm used to train on this data are both chosen as .1. By tracking the test set error while training with these parameters, an adequate training time is determined to be 63 epochs. The classifier performance at this point is 6.8% on the training set and 11% on the test set. Subsequently, the feature set is reduced, using forward sequential selection,

to 34 input features. Retraining on this reduced data set results in a final error on the training and test sets of .6% and 5.5%, respectively. For evaluation purposes, the upper bound on Bayes error is computed over ten hidden layer nodes. The upper bound at five hidden layer nodes, which is the number of hidden layer nodes used on all classifiers in this research, is 11%.

*4.3.2 Actinide.* AFTAC provided a single actinide data set labeled with alphanumeric descriptors representing the locations from which the samples were taken. By allowing each descriptor to represent a class, 55 classes result. Because many of the classes contain very few samples, the class structure is altered to allow only 15 larger classes. Therefore, the two actinide data sets are identical with the exception of the class structure. Each contains 18 input features.

*4.3.2.1 55-class Actinide.* As with the steel data, the training parameters are selected using Fallside plots. A learning rate of .1 and a momentum of .5 are chosen. The data is divided into training and test sets and a classifier is trained using these parameters. The training set error approaches 57%, while the test set error approaches 59%. Because this level of error is unacceptable for real-world application and it is not clear that further training will improve performance, no additional analysis is conducted with the exception of bounding Bayes error. The upper bound on Bayes error for this data set is 63% (at 5 hidden layer nodes).

*4.3.2.2 15-class Actinide.* The training parameters for this data set are a learning rate and momentum of .1. Training with these parameters on the full feature set results in a training error of 21% and a test error of 26%. During forward sequential selection, the classifiers are trained for 3,000 epochs each and the feature set is reduced from 18 to 9 features. Upon retraining, the test set error is unchanged, while the training set error increases to 24%. The Bayes error upper bound at five hidden layer nodes is found to be 29%.

## *4.4 Conclusions*

The following conclusions may drawn from this thesis:

- The multi-layer perceptron is capable of classifying different type of stainless steel samples with a high degree of accuracy.
- The multi-layer perceptron performs as well as any other classifier when classifying actinide by location, given the data set provided. The overall accuracy of the multi-layer perceptron is unacceptably low for both the 55-class and the 15-class structures. However, the error rates are within the upper bound of Bayes error in both cases. Furthermore, the poor performance is linked to the number of classes containing few samples. On the classes containing an adequate number of samples, the multi-layer perceptron performed well.
- The number of features used in classifying the given environmental data can be significantly reduced with little or no decrease in the classification accuracy.

Considering the results and conclusions presented in this chapter and the **methodology** outlined throughout this work, this thesis has met all of the objectives set forth in Section 1.4.

## *4.5 Recommendations for Follow-on Research*

The possibilities for additional research related to this thesis are:

1. Develop training algorithms capable of minimizing by-class error and overall error simultaneously.
2. For the actinide data, gather more samples in the less populated classes and perform the analysis outlined in this research.
3. Measure alternative features for the actinide data and perform the analysis outlined.

## *Appendix A. Backpropagation Learning Law Derivations*

### *A.1 General Learning Law Derivation*

For the MLP to be trained to classify, a general learning law for each set of weights can be derived independent of the type of transformation function applied at the nodes. Backpropagation is the most common technique, and the one used here, to update the matrix of weights, $\mathbf{W}$. Backpropagation requires that the partial derivative of the error, $E$, be computed with respect to each weight. A widely accepted measurement of the error is the sum squared error, defined by:

$$E = \frac{1}{2}\sum_{k=1}^{K}(d_k - f_k)^2, \tag{A.1}$$

where $d_k$ is the desired output and $f_k$ is the actual output. If $\frac{\partial E}{\partial \mathbf{W}}$ describes a matrix whose elements are given as:

$$\left(\frac{\partial E}{\partial \mathbf{W}}\right)_{ij} = \frac{\partial E}{\partial w_{ij}}$$

then the learning law for each set of weights is generally written as:

$$\mathbf{W}^+ = \mathbf{W}^- - \eta\frac{\partial E}{\partial \mathbf{W}^-}$$

where $\mathbf{W}^+$ is the updated weight set, $\mathbf{W}^-$ is the old weight set and $\eta$ is the learning rate.

In the next two sections, the weight update equations for a two-layer perceptron are derived.

*A.1.1 Derivation of Output-Layer Weight Updates.* The update equations for the output layer are considered in this section followed by a section containing the derivation of the update equations for the hidden layer. An output-layer weight is designated by subscripts $j$ and $k$, specifying the connection between the $j^{th}$ node in the hidden layer and the $k^{th}$ node in the output layer. Each output from the output layer is designated by $f_k$. From the generalized form of the learning law, the updated weight is established as:

$$w^{+}_{j_0 k_0} = w^{-}_{j_0 k_0} - \eta \frac{\partial E}{\partial w_{j_0 k_0}}$$

where the error E for the two-layer perceptron is defined as:

$$E = \frac{1}{2} \sum_{k=1}^{K} (d_k - f_k)^2$$

To implement the update equation, the partial derivative of the error, $E$, with respect to the weight $w_{j_0 k_0}$ is evaluated.

$$\frac{\partial E}{\partial w_{j_0 k_0}} = \frac{\partial}{\partial w_{j_0 k_0}} \{ \frac{1}{2} \sum_{k=1}^{K} (d_k - f_k)^2 \}$$

In the partial derivative of the summation from the above equation, the summation's dependency on $w_{j_0 k_0}$ must be isolated.

$$\frac{1}{2} \sum_{k=1}^{K} (d_k - f_k)^2 = \frac{1}{2} \{ (d_1 - f_{k_1})^2 + \cdots + (d_{k_0} - f_{k_0})^2 + \cdots + (d_k - f_k)^2 \}$$

Taking the partial derivative of the equation above, the only part to survive the differentiation will be variables that involve both subscripts $j_0$ and $k_0$. This reasoning identifies the terms $f_{k_0}$ and $d_{k_0}$. While the desired output $d_{k_0}$ is a constant, the actual output $f_{k_0}$ is a function of the summation of weighted outputs from the hidden layer. The partial derivative of the summation simplifies to:

$$\frac{\partial (d_k - f_k)^2}{\partial w_{j_0 k_0}} = \begin{cases} 0 & : \ k \neq k_0 \\ 2(d_{k_0} - f_{k_0}) \dfrac{\partial (-f_{k_0})}{\partial w_{j_0 k_0}} & : \ k = k_0 \end{cases}$$

and therefore:

$$\frac{\partial E}{\partial w_{j_0 k_0}} = \frac{\partial}{\partial w_{j_0 k_0}} \{ \frac{1}{2} \sum_{k=1}^{K} (d_k - f_k)^2 \} = (d_{k_0} - f_{k_0}) \frac{\partial (-f_{k_0})}{\partial w_{j_0 k_0}}$$

As stated above, $f_{k_0}$ is a function of the summation of weighted outputs from the hidden layer. The partial derivative of $f_{k_0}$ can be written in the generalized form as:

$$\frac{\partial f_{k_0}}{\partial w_{j_0 k_0}} = \frac{\partial}{\partial w_{j_0 k_0}} f_{k_0}\Big(\sum_{j=1}^{J+1} w_{jk_0} f_j\Big)$$

where $f_{k_0}(\alpha)$ is the transformation employed at node $k_0$. By substitution:

$$\begin{aligned}
\frac{\partial E}{\partial w_{j_0 k_0}} &= -(d_{k_0} - f_{k_0}) \frac{\partial}{\partial w_{j_0 k_0}} f_{k_0}(\sum_{j=1}^{J+1} w_{jk_0} f_j) \\
&= -(d_{k_0} - f_{k_0}) f'_{k_0} \frac{\partial}{\partial w_{j_0 k_0}} \sum_{j=1}^{J+1} w_{jk_0} f_j \\
&= -(d_{k_0} - f_{k_0}) f'_{k_0} f_{j_0}
\end{aligned}$$

The notation $f'_{k_0}$ represents the derivative of $f_{k_0}(\alpha)$ with respect to $\alpha$. After the differentiation, the argument $\alpha$ is replaced by the activation function $\sum_{j=1}^{J+1} w_{jk_0} f_j$.

Therefore, the output-layer weight update equation reduces to:

$$w^+_{j_0 k_0} = w^-_{j_0 k_0} + \eta(d_{k_0} - f_{k_0}) f'_{k_0} f_{j_0} \tag{A.2}$$

*A.1.2 Derivation of Hidden-Layer Weight Updates.* The hidden-layer weight is designated by subscripts $i$ and $j$, specifying the connection between the $i^{th}$ node in the input layer and the $j^{th}$ node in the hidden layer. Each output from the hidden layer is designated by $f_j$. Similar to the updated weight for the output layer, the hidden-layer updated weight is established as:

$$w^+_{i_0 j_0} = w^-_{i_0 j_0} - \eta \frac{\partial E}{\partial w_{i_0 j_0}}$$

The error term here is identical to that discussed in the last section. To implement the update equation, the partial derivative of the error $E$ must be evaluated with respect to the old weight $w_{i_0 j_0}$ for this layer rather than $w_{j_0 k_0}$.

Because the partial derivative depends only on the links containing the $i_0$ and $j_0$ nodes and because $f_k$ is a function of the summation of weighted outputs $f_j$ from the hidden layer, the partial derivative of the error term is given as:

$$\frac{\partial E}{\partial w_{i_0 j_0}} = \frac{\partial}{\partial w_{i_0 j_0}}\{\frac{1}{2}\sum_{k=1}^{K}(d_k - f_k)^2\} = \sum_{k=1}^{K}(d_k - f_k)\frac{\partial(-f_k)}{\partial w_{i_0 j_0}}$$

Substituting this along with the equation for $f_k$ from the previous section, the following equation for the partial derivative of the error term may be derived:

$$\begin{aligned}
\frac{\partial E}{\partial w_{i_0 j_0}} &= \frac{\partial}{\partial w_{i_0 j_0}}\{\frac{1}{2}\sum_{k=1}^{K}(d_k - f_k)^2\} \\
&= \sum_{k=1}^{K}(d_k - f_k)\frac{\partial(-f_k)}{\partial w_{i_0 j_0}} \\
&= -\sum_{k=1}^{K}(d_k - f_k)\frac{\partial}{\partial w_{i_0 j_0}} f_k\Big(\sum_{j=1}^{J+1} w_{jk} f_j\Big) \\
&= -\sum_{k=1}^{K}(d_k - f_k) f_k' \frac{\partial}{\partial w_{i_0 j_0}} \sum_{j=1}^{J+1} w_{jk} f_j \\
&= -\sum_{k=1}^{K}(d_k - f_k) f_k' w_{j_0 k} \frac{\partial(f_{j_0})}{\partial w_{i_0 j_0}}
\end{aligned}$$

where $f_k'$ represents the derivative of $f_k(\alpha)$ with respect to $\alpha$.

The output of a hidden-layer node is given by $f_{j_0}(\alpha)$ where

$$\alpha = \Big(\sum_{i=1}^{I+1} w_{i j_0} x_i\Big)$$

Substituting the above equation for $f_{j_0}$ and realizing that

$$\frac{\partial(w_{i j_0} x_i)}{\partial w_{i_0 j_0}} = \begin{cases} 0 & : \ i \neq i_0 \\ x_{i_0} & : \ i = i_0 \end{cases}$$

gives

$$\begin{aligned}\frac{\partial E}{\partial w_{i_0 j_0}} &= -\sum_{k=1}^{K}(d_k - f_k)f'_k w_{j_0 k}\frac{\partial}{\partial w_{i_0 j_0}}f_{j_0}\Big(\sum_{i=1}^{I+1} w_{ij_0}x_i\Big) \\ &= -\sum_{k=1}^{K}(d_k - f_k)f'_k w_{j_0 k}f'_{j_0}\frac{\partial}{\partial w_{i_0 j_0}}\sum_{i=1}^{I+1} w_{ij_0}x_i \\ &= -\sum_{k=1}^{K}(d_k - f_k)f'_k w_{j_0 k}f'_{j_0}x_{i_0}\end{aligned}$$

Therefore, the hidden-layer weight update equation reduces to:

$$w^{+}_{i_0 j_0} = w^{-}_{i_0 j_0} + \eta\sum_{k=1}^{K}(d_k - f_k)f'_k w_{j_0 k}f'_{j_0}x_{i_0} \tag{A.3}$$

*A.1.3 Conclusion.* In sum, the general equations for updating the output-layer and hidden-layer weights are provided in Equations A.2 and A.3. In the next section, the weight update equations will be derived for MLP structures tailored for specific transformation functions.

## *A.2 Transformation Function Specific Derivation of Learning Laws*

*A.2.1 Transformation Functions.* In the following section, learning laws for both layers will be derived for the output defined for the sigmoid, tanh, and linear transformations:

- SIGMOID: $f_{k_0} = \frac{1}{1+e^{-\sum_{j=1}^{J+1} w_{jk_0}f_j}}$;
- TANH: $f_{k_0} = \tanh(\sum_{j=1}^{J+1} w_{jk_0}f_j)$;
- LINEAR: $f_{k_0} = \sum_{j=1}^{J+1} w_{jk_0}f_j$.

The combination of transformations to be considered at the output and hidden layers is shown in Table A.1.

Table A.1 Transformation Function Case Table

| Layer | Case | | | |
|---|---|---|---|---|
| Hidden | Sigmoid | Sigmoid | Tanh | Tanh |
| Output | Sigmoid | Linear | Tanh | Linear |

*A.2.2 Case I: Sigmoid-Sigmoid.* For the output layer,

$$w^{+}_{j_0k_0} = w^{-}_{j_0k_0} - \eta \frac{\partial E}{\partial w_{j_0k_0}}$$

Now, just analyzing the partial derivative term in the expression above yields the following:

$$\frac{\partial E}{\partial w_{j_0k_0}} = \frac{\partial}{\partial w_{j_0k_0}}\{\frac{1}{2}\sum_{k=1}^{K}(d_k - f_k)^2\} = (d_{k_0} - f_{k_0})(-1)\frac{\partial f_{k_0}}{\partial w_{j_0k_0}}$$

and substituting the sigmoidal transformation function for $f_{k_0}$ in the derivative yields:

$$\frac{\partial E}{\partial w_{j_0k_0}} = -(d_{k_0} - f_{k_0})\frac{\partial}{\partial w_{j_0k_0}}(1 + e^{-\sum_{j=1}^{J+1} w_{jk_0} f_j})^{-1}$$

$$\frac{\partial E}{\partial w_{j_0k_0}} = -(d_{k_0} - f_{k_0})(-1)(1 + e^{-\sum_{j=1}^{J+1} w_{jk_0} f_j})^{-2}(e^{-\sum_{j=1}^{J+1} w_{jk_0} f_j})\frac{\partial}{\partial w_{j_0k_0}}(-\sum_{j=1}^{J+1} w_{jk_0} f_j)$$

$$\frac{\partial E}{\partial w_{j_0k_0}} = -(d_{k_0} - f_{k_0})\frac{e^{-\sum_{j=1}^{J+1} w_{jk_0} f_j}}{(1 + e^{-\sum_{j=1}^{J+1} w_{jk_0} f_j})^2}(f_{j_0})$$

$$\frac{\partial E}{\partial w_{j_0k_0}} = -(d_{k_0} - f_{k_0})(f_{k_0})(1 - f_{k_0})(f_{j_0})$$

Therefore, for the output layer of the Sigmoid-Sigmoid Case:

$$w^{+}_{j_0k_0} = w^{-}_{j_0k_0} + \eta(d_{k_0} - f_{k_0})(f_{k_0})(1 - f_{k_0})(f_{j_0}) \tag{A.4}$$

Now, the hidden layer weights must be updated.

$$w^{+}_{i_0j_0} = w^{-}_{i_0j_0} - \eta\frac{\partial E}{\partial w_{i_0j_0}}$$

The partial derivative term of the above expression will be analyzed as before.

$$\frac{\partial E}{\partial w_{i_0j_0}} = \frac{\partial}{\partial w_{i_0j_0}}\{\frac{1}{2}\sum_{k=1}^{K}(d_k - f_k)^2\} = -\sum_{k=1}^{K}(d_k - f_k)\frac{\partial f_k}{\partial w_{i_0j_0}}$$

substituting the sigmoidal transformation function for $f_k$ in the derivative yields:

$$\frac{\partial E}{\partial w_{i_0j_0}} = -\sum_{k=1}^{K}(d_k - f_k)\frac{\partial}{\partial w_{i_0j_0}}(1 + e^{-\sum_{j=1}^{J+1} w_{jk}f_j})^{-1}$$

$$\frac{\partial E}{\partial w_{i_0j_0}} = -\sum_{k=1}^{K}(d_k - f_k)(f_k)(1 - f_k)\frac{\partial}{\partial w_{i_0j_0}}(-\sum_{j=1}^{J+1} w_{jk}f_j)$$

$$\frac{\partial E}{\partial w_{i_0j_0}} = -\sum_{k=1}^{K}(d_k - f_k)(f_k)(1 - f_k)(-w_{j_0k})\frac{\partial}{\partial w_{i_0j_0}}(f_{j_0})$$

substituting the sigmoidal transformation function for $f_{j_0}$ in the derivative and evaluating it yields:

$$\frac{\partial E}{\partial w_{i_0j_0}} = -\sum_{k=1}^{K}(d_k - f_k)(f_k)(1 - f_k)(-w_{j_0k})(f_{j_0})(1 - f_{j_0})(-x_{i_0})$$

Therefore, for the hidden layer of the Sigmoid-Sigmoid Case:

$$w^+_{i_0 j_0} = w^-_{i_0 j_0} + \eta \sum_{k=1}^{K} (d_k - f_k)(f_k)(1 - f_k)(-w_{j_0 k})(f_{j_0})(1 - f_{j_0})(x_{i_0}) \qquad \text{(A.5)}$$

*A.2.3 Case II: Sigmoid-Linear.* For the output layer,

$$w^+_{j_0 k_0} = w^-_{j_0 k_0} - \eta \frac{\partial E}{\partial w_{j_0 k_0}}$$

Now, just analyzing the partial derivative term in the expression above yields the following.

$$\frac{\partial E}{\partial w_{j_0 k_0}} = \frac{\partial}{\partial w_{j_0 k_0}} \frac{1}{2} \sum_{k=1}^{K} (d_k - f_k)^2 = (d_{k_0} - f_{k_0})(-1) \frac{\partial f_{k_0}}{\partial w_{j_0 k_0}}$$

and substituting the linear transformation function for $f_{k_0}$ in the derivative yields:

$$\frac{\partial E}{\partial w_{j_0 k_0}} = -(d_{k_0} - f_{k_0}) \frac{\partial}{\partial w_{j_0 k_0}} \sum_{j=1}^{J+1} w_{j k_0} f_j$$

$$\frac{\partial E}{\partial w_{j_0 k_0}} = -(d_{k_0} - f_{k_0})(f_{j_0})$$

Therefore, for the output layer of the Sigmoid-Linear Case:

$$w^+_{j_0 k_0} = w^-_{j_0 k_0} + \eta (d_{k_0} - f_{k_0})(f_{j_0}) \qquad \text{(A.6)}$$

The hidden layer weights must be updated as shown below.

$$w^+_{i_0 j_0} = w^-_{i_0 j_0} - \eta \frac{\partial E}{\partial w_{i_0 j_0}}$$

The partial derivative term of the above expression will be analyzed as before.

$$\frac{\partial E}{\partial w_{i_0 j_0}} = \frac{\partial}{\partial w_{i_0 j_0}}\{\frac{1}{2}\sum_{k=1}^{K}(d_k - f_k)^2\} = -\sum_{k=1}^{K}(d_k - f_k)\frac{\partial f_k}{\partial w_{i_0 j_0}}$$

and substituting the linear transformation function for $f_k$ in the derivative yields:

$$\frac{\partial E}{\partial w_{i_0 j_0}} = -\sum_{k=1}^{K}(d_k - f_k)\frac{\partial}{\partial w_{i_0 j_0}}\{\sum_{j=1}^{J+1} w_{jk} f_j\}$$

$$\frac{\partial E}{\partial w_{i_0 j_0}} = -\sum_{k=1}^{K}(d_k - f_k)(w_{j_0 k})\frac{\partial}{\partial w_{i_0 j_0}}(f_{j_0})$$

substituting the sigmoidal transformation function for $f_{j_0}$ in the derivative and evaluating it yields:

$$\frac{\partial E}{\partial w_{i_0 j_0}} = -\sum_{k=1}^{K}(d_k - f_k)(w_{j_0 k})(f_{j_0})(1 - f_{j_0})(x_{i_0})$$

Therefore, for the hidden layer of the Sigmoid-Linear Case:

$$w^{+}_{i_0 j_0} = w^{-}_{i_0 j_0} + \eta\sum_{k=1}^{K}(d_k - f_k)(w_{j_0 k})(f_{j_0})(1 - f_{j_0})(x_{i_0}) \qquad \text{(A.7)}$$

*A.2.4 Case III: Tanh-Tanh.* For, the output layer,

$$w^{+}_{j_0 k_0} = w^{-}_{j_0 k_0} - \eta\frac{\partial E}{\partial w_{j_0 k_0}}$$

Just analyzing the partial derivative term in the expression above yields the following.

$$\frac{\partial E}{\partial w_{j_0 k_0}} = \frac{\partial}{\partial w_{j_0 k_0}}\{\frac{1}{2}\sum_{k=1}^{K}(d_k - f_k)^2\} = (d_{k_0} - f_{k_0})(-1)\frac{\partial f_{k_0}}{\partial w_{j_0 k_0}}$$

and substituting the hyperbolic tangent transformation function for $f_{k_0}$ in the derivative yields:

$$\frac{\partial E}{\partial w_{j_0 k_0}} = -(d_{k_0} - f_{k_0}) \frac{\partial}{\partial w_{j_0 k_0}} \{\tanh \sum_{j=1}^{J+1} w_{j k_0} f_j\}$$

$$\frac{\partial E}{\partial w_{j_0 k_0}} = -(d_{k_0} - f_{k_0})(\cosh \sum_{j=1}^{J+1} w_{j k_0} f_j)^{-2} \frac{\partial}{\partial w_{j_0 k_0}} \{\sum_{j=1}^{J+1} w_{j k_0} f_j\}$$

$$\frac{\partial E}{\partial w_{j_0 k_0}} = -(d_{k_0} - f_{k_0})(\cosh \sum_{j=1}^{J+1} w_{j k_0} f_j)^{-2} (f_{j_0})$$

$$w^+_{j_0 k_0} = w^-_{j_0 k_0} + \eta (d_{k_0} - f_{k_0})(\cosh \sum_{j=1}^{J+1} w_{j k_0} f_j)^{-2} (f_{j_0})$$

Therefore, for the output layer of the Tanh-Tanh Case:

$$w^+_{j_0 k_0} = w^-_{j_0 k_0} + \eta (d_{k_0} - f_{k_0})(1 - (f_{k_0})^2)(f_{j_0}) \tag{A.8}$$

The hidden layer weights must be updated as shown below.

$$w^+_{i_0 j_0} = w^-_{i_0 j_0} - \eta \frac{\partial E}{\partial w_{i_0 j_0}}$$

The partial derivative term of the above expression will be analyzed as before.

$$\frac{\partial E}{\partial w_{i_0 j_0}} = \frac{\partial}{\partial w_{i_0 j_0}} \{\frac{1}{2} \sum_{k=1}^{K} (d_k - f_k)^2\} = -\sum_{k=1}^{K} (d_k - f_k) \frac{\partial f_k}{\partial w_{i_0 j_0}}$$

substituting the hyperbolic tangent transformation function for $f_k$ in the derivative and evaluating it yields:

$$\frac{\partial E}{\partial w_{i_0 j_0}} = -\sum_{k=1}^{K}(d_k - f_k)(\cosh \sum_{j=1}^{J+1} w_{jk} f_j)^{-2} \frac{\partial}{\partial w_{i_0 j_0}} \sum_{j=1}^{J+1} w_{jk} f_j$$

$$\frac{\partial E}{\partial w_{i_0 j_0}} = -\sum_{k=1}^{K}(d_k - f_k)(\cosh \sum_{j=1}^{J+1} w_{jk} f_j)^{-2} \frac{\partial}{\partial w_{i_0 j_0}} \sum_{j=1}^{J+1} w_{jk} f_j$$

$$\frac{\partial E}{\partial w_{i_0 j_0}} = -\sum_{k=1}^{K}(d_k - f_k)(\cosh \sum_{j=1}^{J+1} w_{jk} f_j)^{-2} (w_{j_0 k}) \frac{\partial}{\partial w_{i_0 j_0}} (f_{j_0})$$

substituting the hyperbolic tangent transformation function for $f_{j_0}$ in the derivative yields:

$$\frac{\partial E}{\partial w_{i_0 j_0}} = -\sum_{k=1}^{K}(d_k - f_k)(\cosh \sum_{j=1}^{J+1} w_{jk} f_j)^{-2} (w_{j_0 k})(\cosh \sum_{i=1}^{I+1} w_{i j_0} x_i)^{-2} \frac{\partial}{\partial w_{i_0 j_0}} \{\sum_{i=1}^{I+1} w_{i j_0} x_i\}$$

$$\frac{\partial E}{\partial w_{i_0 j_0}} = -\sum_{k=1}^{K}(d_k - f_k)(\cosh \sum_{j=1}^{J+1} w_{jk} f_j)^{-2} (w_{j_0 k})(\cosh \sum_{i=1}^{I+1} w_{i j_0} x_i)^{-2} (x_{i_0})$$

$$w^{+}_{i_0 j_0} = w^{-}_{i_0 j_0} + \eta \sum_{k=1}^{K}(d_k - f_k)(\cosh \sum_{j=1}^{J+1} w_{jk} f_j)^{-2} (w_{j_0 k})(\cosh \sum_{i=1}^{I+1} w_{i j_0} x_i)^{-2} (x_{i_0})$$

Therefore, for the hidden layer of the Tanh-Tanh Case:

$$w^{+}_{i_0 j_0} = w^{-}_{i_0 j_0} + \eta \sum_{k=1}^{K}(d_k - f_k)(1 - (f_k)^2)(w_{j_0 k})(1 - (f_{j_0})^2)(x_{i_0}) \qquad \text{(A.9)}$$

*A.2.5 Case IV: Tanh-Linear.* For the output layer, the learning law is the same as the learning law derived earlier in Case II (Equation A.6).

$$w^{+}_{j_0 k_0} = w^{-}_{j_0 k_0} + \eta(d_{k_0} - f_{k_0})(f_{j_0})$$

For the hidden layer weights the derivation is provided below.

$$w^{+}_{i_0 j_0} = w^{-}_{i_0 j_0} - \eta \frac{\partial E}{\partial w_{i_0 j_0}}$$

The partial derivative term of the expression above is analyzed below.

$$\frac{\partial E}{\partial w_{i_0 j_0}} = \frac{\partial}{\partial w_{i_0 j_0}} \{\frac{1}{2} \sum_{k=1}^{K} (d_k - f_k)^2\} = -\sum_{k=1}^{K} (d_k - f_k) \frac{\partial f_k}{\partial w_{i_0 j_0}}$$

substituting the linear transformation function for $f_k$ in the derivative, evaluating it, and substituting the hyperbolic tangent transformation function in the resulting derivative for $f_{j_0}$ yields:

$$\frac{\partial E}{\partial w_{i_0 j_0}} = -\sum_{k=1}^{K} (d_k - f_k)(w_{j_0 k}) \frac{\partial}{\partial w_{i_0 j_0}} \{\tanh \sum_{i=1}^{I+1} w_{i j_0} x_i\}$$

$$\frac{\partial E}{\partial w_{i_0 j_0}} = -\sum_{k=1}^{K} (d_k - f_k)(w_{j_0 k})(\cosh \sum_{i=1}^{I+1} w_{i j_0} x_i)^{-2}(x_{i_0})$$

Therefore, for the hidden layer of the Tanh-Linear Case:

$$w^{+}_{i_0 j_0} = w^{-}_{i_0 j_0} + \eta \sum_{k=1}^{K} (d_k - f_k)(w_{j_0 k})(1 - (f_{j_0})^2)(x_{i_0}) \qquad \text{(A.10)}$$

# *Appendix B. Matlab Code*

## *B.1 removeh.m*

```
% This function removes the homogenous features in the data set
% "datain". The reduced data set is "dataout". The features
% that were deleted are stored in "featsdel", while the the
% features that were kept are in "featskept".
%       syntax:  [dataout featsdel featskept]=removeh(datain);

function [dataout,featsdel,featskept] = removeh(datain)
[row col]=size(datain);
target=datain(1,:);
featsdel==zeros(1,col);
featskept==zeros(1,col);
for j=2:col,
   temp=find(datain(:,j)~=target(1,j));
   if isempty(temp),
      featsdel(1,j)=1;
   else
      featskept(1,j)=1;
   end;
end;
index=find(featskept);
dataout=[datain(:,1) datain(:,index)];
```

## *B.2 normal.m*

```
% This function performs a simple normalization of the
% data "datain".
%       syntax:  dataout=normal(datain);
function [datain]= normal(datain)
[row col]=size(datain);
means=mean(datain);
stds=std(datain);
for i=1:col
```

```
   if (stds(1,i)==0),
      stds(1,i)=1e-15;
   end;
end;
for i=1:row,
   datain(i,:)=(datain(i,:)-means)./stds;
end;
```

### *B.3 randchoose.m*

```
% This function divides the data set "datain" into two
% subsets "train" and "test" by randomly selecting
% members by class for each subset.
%        syntax:  [train test]=randchoose(train)

function [train,test]=randchoose(datain)
train=[];
test=[];
[row col]=size(datain);
Order=randperm(row);
datain=datain(Order,:);
for i=1:max(datain(:,1)),
   index=find(datain(:,1)==i);
   N=size(index,1);
   if N==1,
      train=[train;datain(index,:)];
      test=[test;datain(index,:)];
   elseif N==2,
      train=[train;datain(index(1),:)];
      test=[test;datain(index(2),:)];
   else
      interval=2*fix(N/3);
      train=[train;datain(index(1:interval),:)];
      test=[test;datain(index(interval+1:N),:)];
   end;
```

```
end;
```

### *B.4 fallside.m*

```
% This function creates fallside plots.
%    sytntax: [x y] = fallside(datain,fid,J,K,f1,f2,l,m,df,me);
% x - matrix containing the classification error curves
%   y - matrix containing the SSE curves
% datain - matrix containing the input data
%  fid - file id for the log file (fid=1 writes to standard output)
% J - number of hidden nodes for the neural network to be used
% K - number of output nodes for the neural network to be used
% (K is usually equal to the number of classes in the data)
% f1 - hidden layer transfer function
% f2 - output layer transfer function
% l - vector containing the initial lr, the lr increment, and the final lr
% m - vector containing the initial mc, the mc increment, and the final mc
% df - number of epochs between writes to the log file
% me - max number of epochs for training
function [cecurves,ssecurves] = fallside(datain,fid,J,K,f1,f2,l,m,df,me);

% Initialize the expected output matrix,t, and the min,max matrix P
t=[];
P=[];

%  Create the t matrix based on the data classes in column 1 of the data matrix
   t=zeros(datain(size(datain,1),1),size(datain,1));
   for i=1:size(datain,1),
       t(datain(i,1),i)=1;
   end;

%  Remove the class column from the input data matrix
datain=datain(:,2:size(datain,2));

%  Transpose the matrix to for use with Matlab neural network functions
```

```
datain=datain';

%  Initalize the neural network architecture
I=size(datain,1);  %Number of input nodes

% Initialize the errror vectors for training, testing and evaluation classification error
cecurves=[];
ssecurves=[];

% Initialize the weight and bias matrices
W1=feval(f1,rand(J,I));
W2=feval(f2,rand(K,J));
B1=feval(f1,rand(J,1));
B2=feval(f2,rand(K,1));
W1old=W1;
W2old=W2;
B1old=B1;
B2old=B2;
% Let learning rate and momentum vary from .1 to
% .9 in .2 increments
for lr=l(1):l(2):l(3),
   for mc=m(1):m(2):m(3),
      %  Set input parameters for backpropagation learning
      TP=[df me 0 lr 1 1 mc 1.04];
      %  Train the neural network
      [W1 B1 W2 B2 TE TR CE]=bpm(fid,W1,B1,f1,...
      W2,B2,f2,datain,t,TP);
      TR=TR(1,:);
      %  Augment the ce and sse vectors
      cecurves=[cecurves;[lr mc CE]];
      ssecurves=[ssecurves;[lr mc TR]];
      %  Reset the biases and weights to the original intial
      %  values for the next pass
      W1=W1old;
```

```
      W2=W2old;
      B1=B1old;
      B2=B2old;
      fprintf('\n');
   end;
end;
```

### *B.5 postf.m*

```
function postf(datain,frame,ptype)
for i=1:5:25,
   ep3d2dxy(0:1000,.1:.2:.9,datain(i:i+4,3:1003)...
   ,'xlabel','ylabel',...
   'zlabel',frame,ptype);
end;
```

### *B.6 bpm.m*

```
function [w1,b1,w2,b2,i,tr,ce] = bpm(fid,w1,b1,f1,w2,b2,f2,p,t,tp)
%  This funtion trains a MLP via backpropagation with momentum.
%  The inputs are as follows:
% w1:  Matrix of hidden layer weights
%   b1:  Matrix of hidden layer bias weights
% f1:  String variable denoting hidden layer output function
% w2:  Matrix of output layer weights
% b2:  Matrix of output layer bias weights
% f2:  String variable denoting output layer output function
% p:   Matrix of input data
% t:   Matrix containing the desired output vector for each
%                    input vector
% tp:  Vector containing the input parameters for training
%  The outputs are as follows:
% w1:  Hidden layer weights after training
% b1:  Hidden layer bias weights after training
% w2:  Output layer weights after training
```

```
% b2:  Output layer bias weights after training
%  Note - these parameters may be used to classify data using the "simuff" command
% i:   Number of epochs trained
% tr:  Matrix of sum sqared error during training
% ce:  Vector containing classification error during training
%  Note - this function is a modfied version of the "trainbpx" function which is
%         part of the neural network toolbox.  It has been modified to include
%         classification error calculations
% TRAINING PARAMETERS
df = tp(1); % Number of epochs between displays
me = tp(2); % Maximum number of epochs
eg = tp(3); % Threshold Error
lr = tp(4); % Learning rate or eta
im = tp(5); % Learning rate and Momentum adaptation parameter - not used in this version
dm = tp(6); % Learning rate and Momentum adaptation parameter - not used in this version
mc = tp(7); % Momentum or alpha
er = tp(8); % Error ratio - not used in this version


%  Determine the delta functions for each layer
df1 = feval(f1,'delta');
df2 = feval(f2,'delta');

%  Initalize the weight changes to zero
dw1 = w1*0; db1 = b1*0;
dw2 = w2*0; db2 = b2*0;
%  Initialize the classification error vector
ce=[];

%  Calculate the initial network output
a1 = feval(f1,w1*p,b1);
a2 = feval(f2,w2*a1,b2);

%  Calculate the initial sum squared error and classification error
```

```
e = t-a2;
[temp It]=max(t);
[temp Ia2]=max(a2);
I=find(It==Ia2);
ce=[ce (1-size(I,2)/size(t,2))];
SSE = sumsqr(e);

% Initialize the training record
tr = zeros(2,me+1);
tr(1:2,1) = [SSE; lr];

% Print the ouput message
message = sprintf('TRAINBPX: %%g/%g epochs, lr = %%g, mc=%%g SSE = %%g\n',me);

fprintf(fid,message,0,lr,mc,SSE);

% BACKPROPAGATION PHASE
% Calculate error derivatives
d2 = feval(df2,a2,e);
d1 = feval(df1,a1,d2,w2);
for i=1:me
  % CHECK PHASE
  if SSE < eg, i=i-1; break, end
  % LEARNING PHASE
  % Calculate the weight changes for each layer according
  % to the backpropagation learning rule
  [dw1] = learnbpm(p,d1,lr,mc,dw1);
  [dw2] = learnbpm(a1,d2,lr,mc,dw2);
  new_w1 = w1 + dw1; new_b1 = b1;
  new_w2 = w2 + dw2; new_b2 = b2;
  % PRESENTATION PHASE
  % Calculate the network output and error
  new_a1 = feval(f1,new_w1*p,new_b1);
  new_a2 = feval(f2,new_w2*new_a1,new_b2);
```

```
  new_e = t-new_a2;
  new_SSE = sumsqr(new_e);
  w1 = new_w1; b1 = new_b1; a1 = new_a1;
  w2 = new_w2; b2 = new_b2; a2 = new_a2;
  [temp It]=max(t);
  [temp Ia2]=max(a2);
  I=find(It==Ia2);
  ce=[ce (1-size(I,2)/size(t,2))];
  e = new_e; SSE = new_SSE;
  % BACKPROPAGATION PHASE
  % Calculate the derivative of the error
  d2 = feval(df2,a2,e);
  d1 = feval(df1,a1,d2,w2);
  % TRAINING RECORD
  tr(:,i+1) = [SSE; lr];
  if rem(i,df) == 0
    fprintf(fid,message,i,lr,mc,SSE);
  end
end;
% TRAINING RECORD
tr = tr(1:2,1:(i+1));
if rem(i,df) ~= 0
  fprintf(fid,message,i,lr,SSE);
end
```

### *B.7 bpmte.m*

```
function [w1,b1,w2,b2,i,tr,ce,tsce] = bpm(fid,w1,b1,f1,...
w2,b2,f2,p,p2,t,t2,tp)
%  This funtion trains a MLP via backpropagation with momentum.
%  The inputs are as follows:
% w1:  Matrix of hidden layer weights
%   b1:  Matrix of hidden layer bias weights
% f1:  String variable denoting hidden layer output function
% w2:  Matrix of output layer weights
```

```
% b2:  Matrix of output layer bias weights
% f2:  String variable denoting output layer output function
% p:   Matrix of input data
% t:   Matrix containing the desired output vector for
%                   each input vector
% tp:  Vector containing the input parameters for training
%  The outputs are as follows:
% w1:  Hidden layer weights after training
% b1:  Hidden layer bias weights after training
% w2:  Output layer weights after training
% b2:  Output layer bias weights after training
%  Note - these parameters may be used to classify data using the "simuff" command
% i:   Number of epochs trained
% tr:  Matrix of sum sqared error during training
% ce:  Vector containing classification error during training
%  Note - this function is a modfied version of the "trainbpx" function which is
%         part of the neural network toolbox.  It has been modified to include
%         classification error calculations
% TRAINING PARAMETERS
df = tp(1); % Number of epochs between displays
me = tp(2); % Maximum number of epochs
eg = tp(3); % Threshold Error
lr = tp(4); % Learning rate or eta
im = tp(5); % Learning rate and Momentum adaptation parameter - not used in this version
dm = tp(6); % Learning rate and Momentum adaptation parameter - not used in this version
mc = tp(7); % Momentum or alpha
er = tp(8); % Error ratio - not used in this version

%  Determine the delta functions for each layer
df1 = feval(f1,'delta');
df2 = feval(f2,'delta');

%  Initalize the weight changes to zero
dw1 = w1*0; db1 = b1*0;
```

```
dw2 = w2*0; db2 = b2*0;
% Initialize the classification error vectors for training, test, and eval
ce=[];
tsce=[];

% Calculate the initial network output
a1 = feval(f1,w1*p,b1);
a2 = feval(f2,w2*a1,b2);

% Calculate the initial sum squared error and classification error
e = t-a2;
[temp It]=max(t);
[temp Ia2]=max(a2);
I=find(It==Ia2);
ce=[ce (1-size(I,2)/size(t,2))];
[tsa1 tsa2]=simuff(p2,w1,b1,f1,w2,b2,f2);
[temp It]=max(t2);
[temp Ia]=max(tsa2);
I=find(It==Ia);
tsce=[tsce (1-size(I,2)/size(t2,2))];
SSE = sumsqr(e);

% Initialize the training record
tr = zeros(2,me+1);
tr(1:2,1) = [SSE; lr];

% Print the ouput message
message = sprintf('TRAINBPX: %%g/%g epochs, lr = %%g, mc=%%g SSE = %%g trce=%%3.1f tsce=%%3.1f\n'

fprintf(fid,message,0,lr,mc,SSE,ce(1)*100,tsce(1)*100);
% BACKPROPAGATION PHASE
% Calculate error derivatives
d2 = feval(df2,a2,e);
d1 = feval(df1,a1,d2,w2);
```

```
for i=1:me
  % CHECK PHASE
  if SSE < eg, i=i-1; break, end;
  % LEARNING PHASE
  % Calculate the weight changes for each layer according
  % to the backpropagation learning rule
  [dw1] = learnbpm(p,d1,lr,mc,dw1);
  [dw2] = learnbpm(a1,d2,lr,mc,dw2);
  new_w1 = w1 + dw1; new_b1 = b1;
  new_w2 = w2 + dw2; new_b2 = b2;
  % PRESENTATION PHASE
  % Calculate the network output and error
  new_a1 = feval(f1,new_w1*p,new_b1);
  new_a2 = feval(f2,new_w2*new_a1,new_b2);
  new_e = t-new_a2;
  new_SSE = sumsqr(new_e);
  w1 = new_w1; b1 = new_b1; a1 = new_a1;
  w2 = new_w2; b2 = new_b2; a2 = new_a2;
  [temp It]=max(t);
  [temp Ia2]=max(a2);
  I=find(It==Ia2);
  ce=[ce (1-size(I,2)/size(t,2))];
  [tsa1 tsa2]=simuff(p2,w1,b1,f1,w2,b2,f2);
  [temp It]=max(t2);
  [temp Ia]=max(tsa2);
  I=find(It==Ia);
  tsce=[tsce (1-size(I,2)/size(t2,2))];
  e = new_e; SSE = new_SSE;
  % BACKPROPAGATION PHASE
  % Calculate the derivative of the error
  d2 = feval(df2,a2,e);
  d1 = feval(df1,a1,d2,w2);
  % TRAINING RECORD
  tr(:,i+1) = [SSE; lr];
```

```
  if rem(i,df) == 0
    fprintf(fid,message,i,lr,mc,SSE,ce(i)*100,tsce(i)*100);
  end
end;
% TRAINING RECORD
tr = tr(1:2,1:(i+1));
if rem(i,df) ~= 0
  fprintf(fid,message,i,lr,mc,SSE,ce(i)*100,tsce(i)*100);
end
```

### *B.8 tev.m*

```
function [TRCEtot,TSCEtot]=tev(train,test,fid,J,K,...
f1,f2,lr,mc,df,me)
%  This function calculates the classification error
%  for the test and evaluation sets during training.
%  This is accomplished numerous times to compute the
%  95% confidence bounds for the mean error.  The
%  number of iterations is specified by "passes"
%
%  syntax :  [x y]=tev(infile,fid,N,J,K,f1,f2,lr,mc,df,me,passes);
% x - matrix containing mean training set classification error
%           curve with 95% confidence bounds
% row 1 - lower bound
% row 2 - mean
% row 3 - upper bound
%   y - matrix containing mean test set classifcation error curve w/95%
%           confidence bounds
% z - matrix containing mean eval set classifcation error curve w/95%
%           confidence bounds
% infile - name of the file to be used for random data selection
%  fid - file id for the log file (fid=1 writes to standard output)
% J - number of hidden nodes for the neural network to be used
% K - number of output nodes for the neural network to be used
% (K is usually equal to the number of classes in the data)
```

```
% f1 - hidden layer transfer function
% f2 - output layer transfer function
% lr - learning rate
% mc - momentum
% df - number of epochs between writes to the log file
% me - max number of epochs for training

% Initialize the test and eval error matrices
TSCEtot=[];
TRCEtot=[];

   % Initialize the desired output matrices for each data set
   t=zeros(train(size(train,1),1),size(train,1));
   t2=zeros(test(size(test,1),1),size(test,1));

   fprintf(fid,'Creating t matrix...\n');
   %  Determine the desired output matrices for each data set
   for i=1:size(train,1),
       t(train(i,1),i)=1;
   end;
   for i=1:size(test,1),
       t2(test(i,1),i)=1;
   end;
   %  Remove the class information from each data set and
   %  transpose each data set
   train=train(:,2:size(train,2));
   train=train';
   test=test(:,2:size(test,2));
   test=test';
   % Initalize neural network parameters
   I=size(train,1);  %Number of input nodes
   W1=feval(f1,rand(J,I));
   W2=feval(f2,rand(K,J));
   B1=feval(f1,rand(J,1));
```

```
B2=feval(f2,rand(K,1));
TP=[df me .02 lr 1 1 mc 1.04];
%  Train the neural network
[W1 B1 W2 B2 TE TR CE TSCE]=bpmte(fid,W1,B1,f1,W2,B2,f2,train,test,t,t2,TP);
save weights W1 B1 W2 B2;
%  Augment the classification error for the current run with those of previous runs
TSCEtot=[TSCEtot;TSCE];
TRCEtot=[TRCEtot;CE];
```

### *B.9 fselct.m*

```
function [feats,Eperfeat]=fselct(datain,fid,numfeats,J,K,f1,f2,...
lr,mc,df,me)
%  This function performs forward sequential feature selection
%
%   syntax:
%  [feats,Eperfeat]=fselct(infile,fid,numfeats,J,K,f1,f2,lr,mc,df,me)
%       feats - vector containing the prioritized features
%   Eperfeat - vector containing the error as each feature is
%                   added to the nucleus
% numfeats - the number of features to select out of the total
% infile - mat filename specifies data to use
%  fid - file id for the log file (fid=1 writes to standard output)
% J - number of hidden nodes
% K - number of output nodes for the neural network to be used
% (K is usually equal to the number of classes in the data)
% f1 - hidden layer transfer function
% f2 - output layer transfer function
% lr - learning rate
```

```
% mc - momentum
% df - number of epochs between writes to the log file
% me - max number of epochs for training

%  Initialize vectors
nucleus=[];
Eperfeat=[];
errors=[];
t=[];
fprintf(fid,'Building t Matrix...\n');
t=zeros(datain(size(datain,1),1),size(datain,1));
for i=1:size(datain,1),
   t(datain(i,1),i)=1;
end;
datain=datain(:,2:size(datain,2));
datain=datain';
[row col]=size(datain);
feats=[];
nucleus=[];
availfeats=1:row;
W2=feval(f2,rand(K,J));
B1=feval(f1,rand(J,1));
B2=feval(f2,rand(K,1));
W1old=[];
W2old=W2;
B1old=B1;
B2old=B2;
fprintf(fid,'Performing feature selection...\n');
while size(feats,2)<numfeats,
  fprintf(fid,'\n');
  errors=[];
  W1=[W1old feval(f1,rand(J,1))];
  W1old=W1;
  for z=1:size(availfeats,2),
```

```
      nucleus=[feats availfeats(1,z)];
      newdata=datain(nucleus,:);
      I=size(newdata,1);  %Number of input nodes
      W1=W1old;
      W2=W2old;
      B1=B1old;
      B2=B2old;
      TP=[df me .02 lr 1 1 mc 1.04];
      [W1 B1 W2 B2 TE TR CE]=bpm(0,W1,B1,f1,W2,B2,f2,newdata,t,TP);
      errors=[errors CE(1,size(CE,2))];
      fprintf(fid,'Nucleus: ');
      fprintf(fid,' %g',nucleus);
      fprintf(fid,' ce= %g \n',CE(1,size(CE,2)));
   end;
   [temp minI]=min(errors);
   Eperfeat=[Eperfeat errors(1,minI)];
   feats=[feats availfeats(1,minI)];
   if minI==size(availfeats,2),
      availfeats=availfeats(1,1:minI-1);
   else,
      availfeats=[availfeats(1,1:minI-1) availfeats(minI+1:size(availfeats,2))];
   end;
end;
```

### *B.10 bbayes.m*

```
function [Upper]=Bbayes(datain,fid,Nodes,K,f1,f2,lr,mc,df,me)
%  This scpript estimates bounds the Bayes Error rate
%   Upper - vector containing upper Bayes bound
% datain - matrix containing the input data
%  fid - file id for the log file (fid=1 writes to standard output)
% Nodes - max number of hidden nodes
% K - number of output nodes for the neural network to be used
% (K is usually equal to the number of classes in the data)
% f1 - hidden layer transfer function
```

```
% f2 - output layer transfer function
% lr - learning rate
% mc - momentum
% df - number of epochs between writes to the log file
% me - max number of epochs for training
%  Initialize vectors
Upper=[];
t=[];
fprintf(fid,'Building t Matrix...\n');
%  Create the t matrix based on the data classes in column 1 of the data matrix
   t=zeros(datain(size(datain,1),1),size(datain,1));
   for i=1:size(datain,1),
       t(datain(i,1),i)=1;
   end;
fprintf(fid,'Normalizing Data...\n');
datain=datain(:,2:size(datain,2));
datain=datain';
I=size(datain,1);  %Number of input nodes

fprintf(fid,'Calculating Upper Bound\n');
B2=feval(f2,rand(K,1));
W1old=[];
W2old=[];
B1old=[];
B2old=B2;
for J=1:Nodes,
   fprintf(fid,'Number of Nodes = %g \n',J);
   misses=0;
   W1=[W1old; feval(f1,rand(1,I))];
   W2=[W2old feval(f2,rand(K,1))];
   B1=[B1old; feval(f1,rand(1,1))];
   B2=B2old;
   W1old=W1;
   W2old=W2;
```

```
   B1old=B1;
   for L=1:size(datain,2);
fprintf(fid,'Sample Left Out = %g \n',L);
newsteel=[datain(:,1:(L-1)) datain(:,(L+1):size(datain,2))];
newt=[t(:,1:(L-1)) t(:,(L+1):size(t,2))];
sample=datain(:,L);
samplet=t(:,L);
W1=W1old;
W2=W2old;
B1=B1old;
B2=B2old;
TP=[df me .02 lr 1 1 mc 1.04];
[W1 B1 W2 B2 TE TR CE]=bpm(fid,W1,B1,f1,W2,B2,f2,...
newsteel,newt,TP);
[a1 a2]=simuff(sample,W1,B1,f1,W2,B2,f2);
[temp It]=max(samplet);
[temp Ia2]=max(a2);
if(It~=Ia2),
   misses=misses+1;
end;
clear W1 W2 B1 B2;
   end;
   Upper=[Upper misses/size(datain,2)];
end;
```

### *B.11 ep3d2dxy.m*

```
function ep3d2dxy(x,y,z,xstr,ystr,zstr,makesqTF,logyTF);
% EP3D2DXY Handy utility for printing 3D data onto 2D plot
% ep3d2dxy(x,y,z,xstr,ystr,zstr,makesqTF,logyTF);
%
% by: Capt Toby Reeves, Capt. Edward M. Ochoa, GEO-96D

x=x(:);
y=y(:);
```

```
figure
if ~logyTF
   h=plot(x,z(1,:)');
   hold on
   for i=2:size(z,1)
      h=[h;plot(x,z(i,:)')];
   end
else
   h=semilogy(x,z(1,:)');
   hold on
   for i=2:size(z,1)
      h=[h;semilogy(x,z(i,:)')];
   end
end
hold off


style=str2mat('-','--',':','-.');
color=str2mat('y','m','c','r','g','b');
slen=size(style,1);
clen=size(color,1);
linewidth=.5;
for i=1:length(h)
  set(h(i),'LineStyle',style(rem(i-1,slen)+1,:))
        if (rem(i,4) | ~rem(i,4)),
linewidth=linewidth+.25;
linecolr=color(rem(i-1,clen)+1,:);
end
  set(h(i),'LineWidth',linewidth)
  set(h(i),'Color',linecolr)
end

ylabel(zstr)
```

```
xlabel(xstr)
%title(sprintf('(%s, %s) vs. %s',xstr,ystr,zstr))
title('title');

if makesqTF
   axis square
end

lgnd=[];
lgnd = num2str(y(1));
for i=2:size(y,1)
   lgnd = str2mat(lgnd,num2str(y(i)));
end

h=legend(lgnd,-1);
axes(h);
title(sprintf('%s',ystr))
```

### *B.12 errorbars.m*

```
function [devs,means]=errorbars(datain,features,classes)
% syntax [devs means]=errorbars(datain,features,classes)
devs=[];
means=[];
for i=1:max(datain(:,1)),
   index=find(datain(:,1)==i);
   if size(index,1)~=1,
      temp=std(datain(index,:));
      devs=[devs;temp];
      means=[means;mean(datain(index,:))];
   else
      devs=[devs;zeros(1,size(datain,2))];
      means=[means;datain(index,:)];
   end;
end;
```

```
for i=1:size(features,2),
   figure;
   for j=1:size(classes,2),
      m=means(classes(1,j),features(1,i)+1);
      s=devs(classes(1,j),features(1,i)+1);
      y=m-s:2*s/10:m+s;
      x=ones(1,size(y,2))*classes(1,j);
      plot(classes(1,j),m-s,'w+');
      hold on;
      plot(classes(1,j),m+s,'w+');
      plot(classes(1,j),m,'r*');
      plot(x,y)
   end;
   xlabel('Class');
   title(['Mean and Standard Deviation of Feature ' int2str(features(1,i)) ...
   ' vs. Class']);
end;
```

### *B.13 removec.m*

```
function [dataout,sizes]=removec(datain,N)
% syntax: [dataout class_sizes]=removec(datain, Target_N_per_class);
dataout=[];
cnt=0;
classes=datain(:,1);
features=datain(:,2:size(datain,2));
sizes=[];
for i=1:max(classes),
index=find(classes==i);
sizes=[sizes size(index,1)];
if size(index,1) > N,
cnt=cnt+1;
    dataout=[dataout;[ones(size(index,1),1)*cnt features(index,:)]];
end;
end;
```

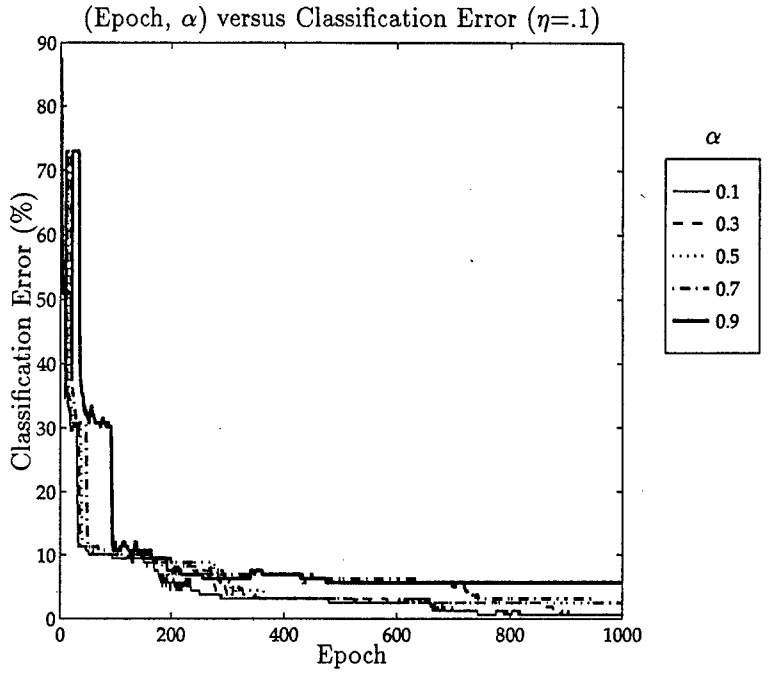
*Appendix C. Elemental Symbols*

| Element | Symbol | Element | Symbol |
|---|---|---|---|
| Lithium | Li | Antimony | Sb |
| Beryllium | Be | Tellurium | Te |
| Boron | B | Iodine | I |
| Carbon | C | Polonium | Po |
| Nitrogen | N | Cesium | Cs |
| Oxygen | O | Barium | Ba |
| Fluorine | F | Lanthanum | La |
| Dysprosium | Dy | Cerium | Ce |
| Sodium | Na | Praseodymium | Pr |
| Magnesium | Mg | Neodymium | Nd |
| Aluminum | Al | Promethium | Pm |
| Silicon | Si | Samarium | Sm |
| Phosphorus | P | Europium | Eu |
| Sulfur | S | Gadolinium | Gd |
| Chlorine | Cl | Terbium | Tb |
| Potassium | K | Holmium | Ho |
| Calcium | Ca | Erbium | Er |
| Scandium | Sc | Thulium | Tm |
| Titanium | Ti | Ytterbium | Yb |
| Vanadium | V | Lutetium | Lu |
| Chromium | Cr | Hafnium | Hf |
| Manganese | Mn | Tantalum | Ta |
| Iron | Fe | Tungsten | W |
| Cobalt | Co | Rhenium | Re |

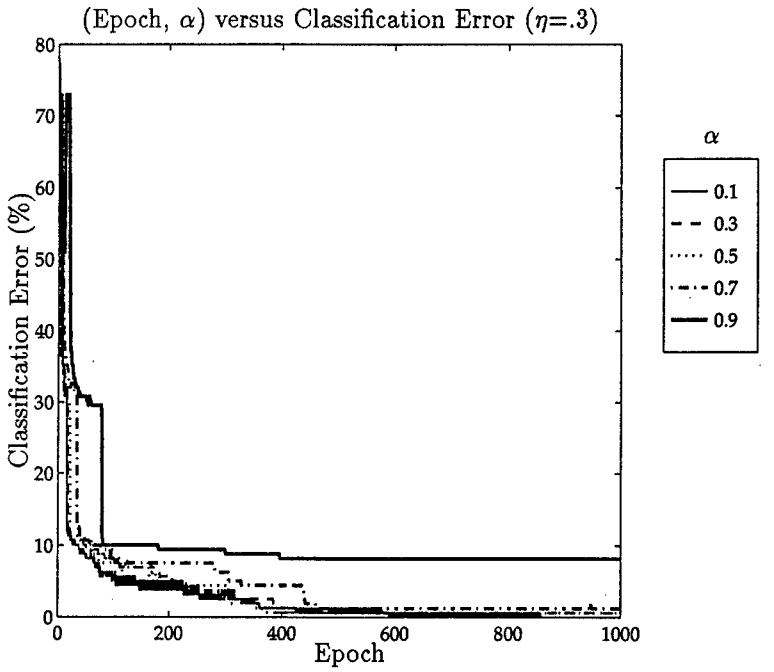
| Element | Symbol | Element | Symbol |
|---|---|---|---|
| Nickel | Ni | Osmium | Os |
| Copper | Cu | Iridium | Ir |
| Zinc | Zn | Platinum | Pt |
| Gallium | Ga | Gold | Au |
| Germanium | Ge | Mercury | Hg |
| Arsenic | As | Thallium | Tl |
| Selenium | Se | Lead | Pb |
| Bromine | Br | Bismuth | Bi |
| Rubidium | Rb | Astatine | At |
| Strontium | Sr | Californium | Cf |
| Yttrium | Y | Francium | Fr |
| Zirconium | Zr | Radium | Ra |
| Niobium | Nb | Actinium | Ac |
| Molybdenum | Mo | Thorium | Th |
| Technetium | Tc | Protactinium | Pa |
| Ruthenium | Ru | Uranium | U |
| Rhodium | Rh | Neptunium | Np |
| Palladium | Pd | Plutonium | Pu |
| Silver | Ag | Americium | Am |
| Cadmium | Cd | Curium | Cm |
| Indium | In | Berkelium | Bk |
| Tin | Sn | | |

# *Appendix D. Fallside Plots*

*D.1 Steel*

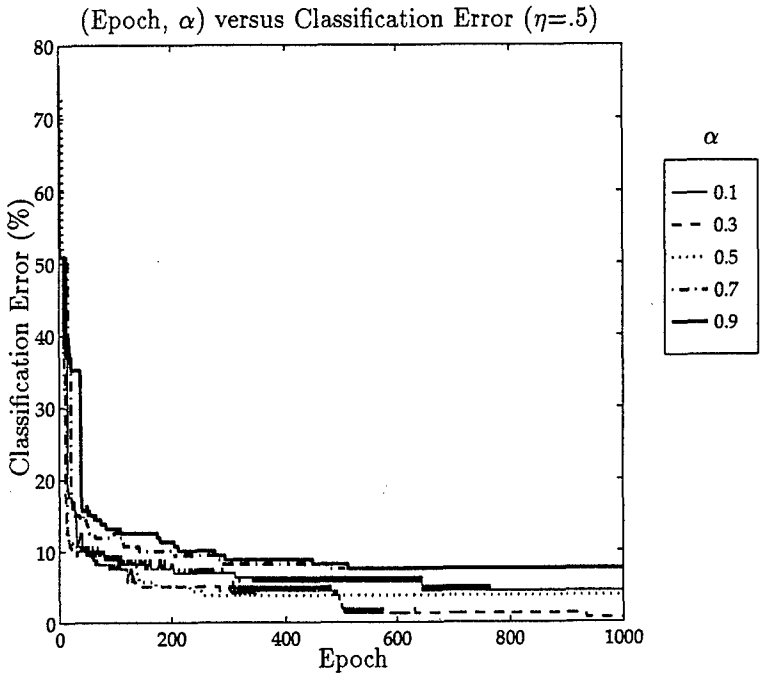



(a)

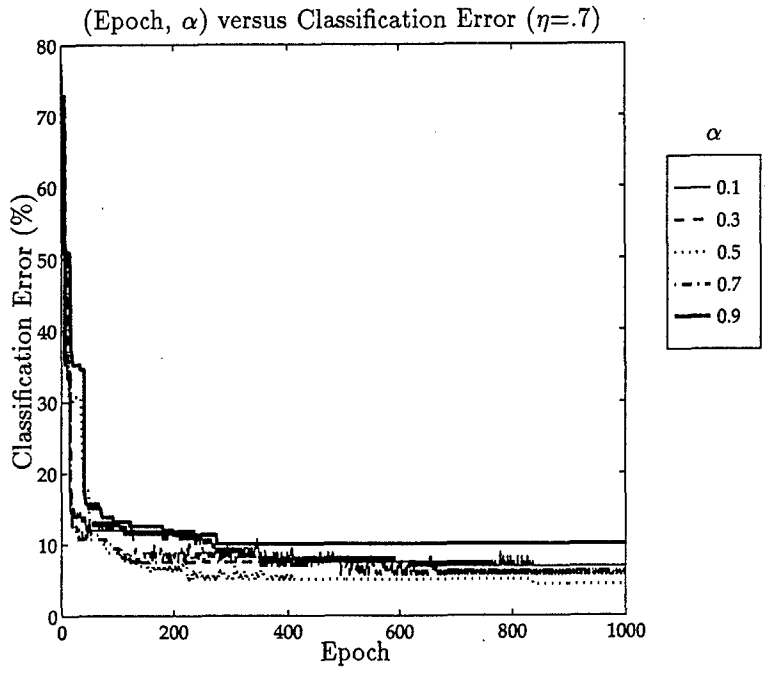



(b)

Figure D.1 *Steel* Fallside plots: classification error versus epoch for (a) $\eta$=.1 and (b)$\eta$=.3 over a range of $\alpha$ values.




(a)




(b)

Figure D.2 *Steel* Fallside plots: classification error versus epoch for (a)$\eta$=.5 and (b)$\eta$=.7 over a range of $\alpha$ values.

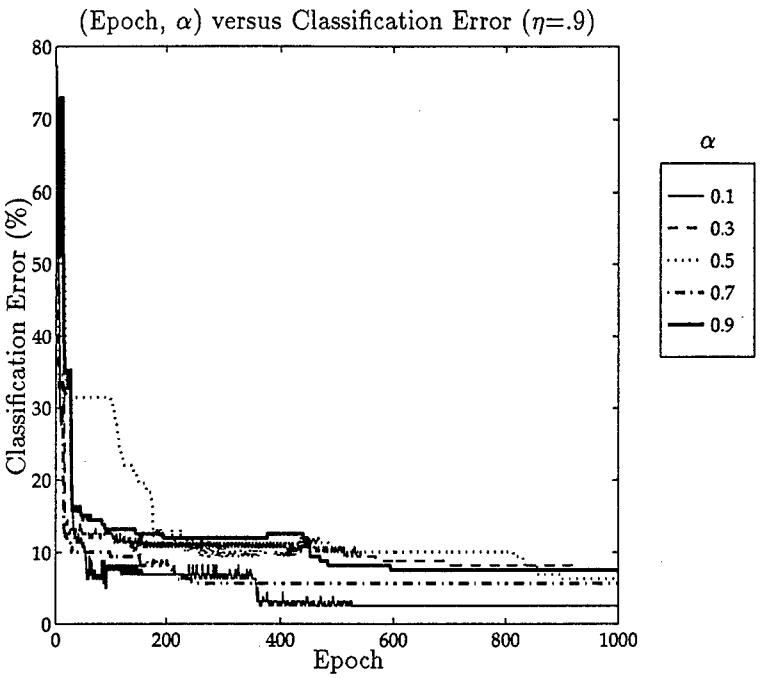
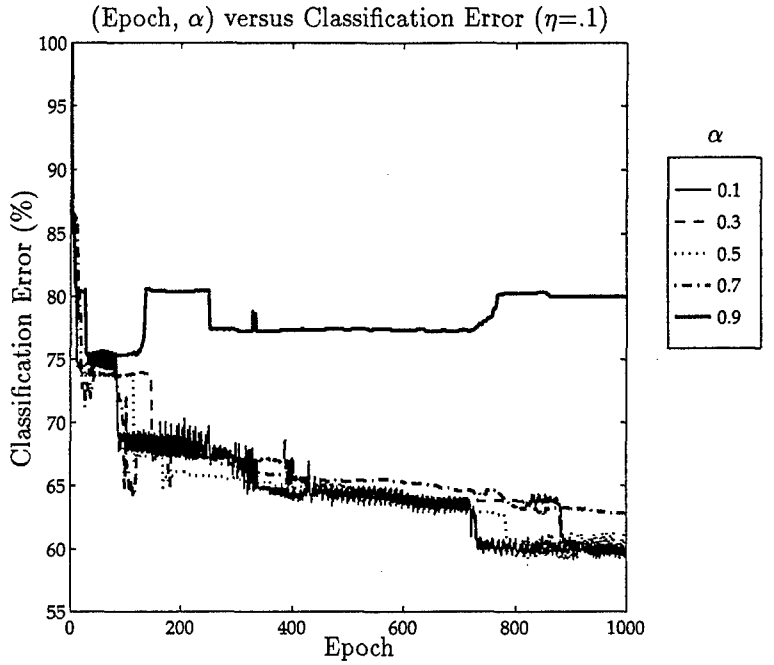
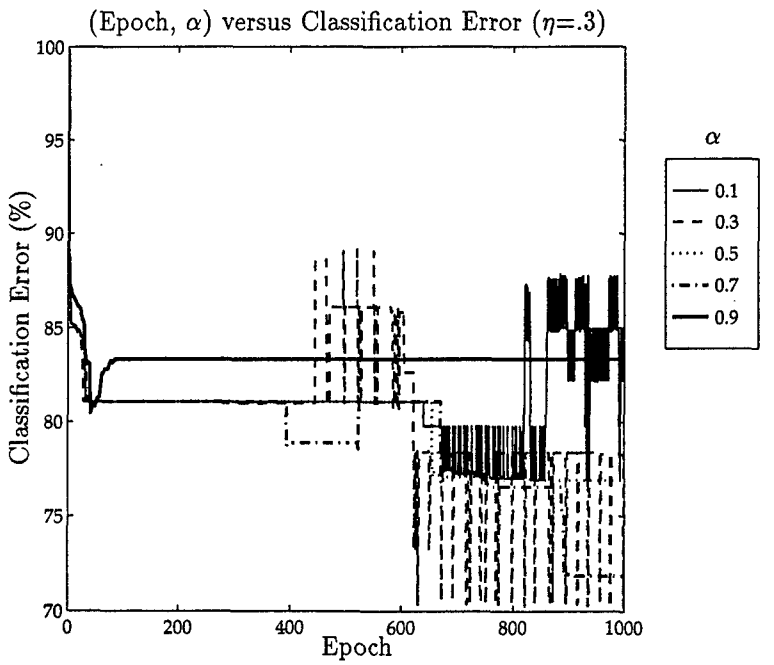



Figure D.3 *Steel* Fallside plots: classification error versus epoch for $\eta$=.9 over a range of $\alpha$ values.

## D.2 $Actinide_1$

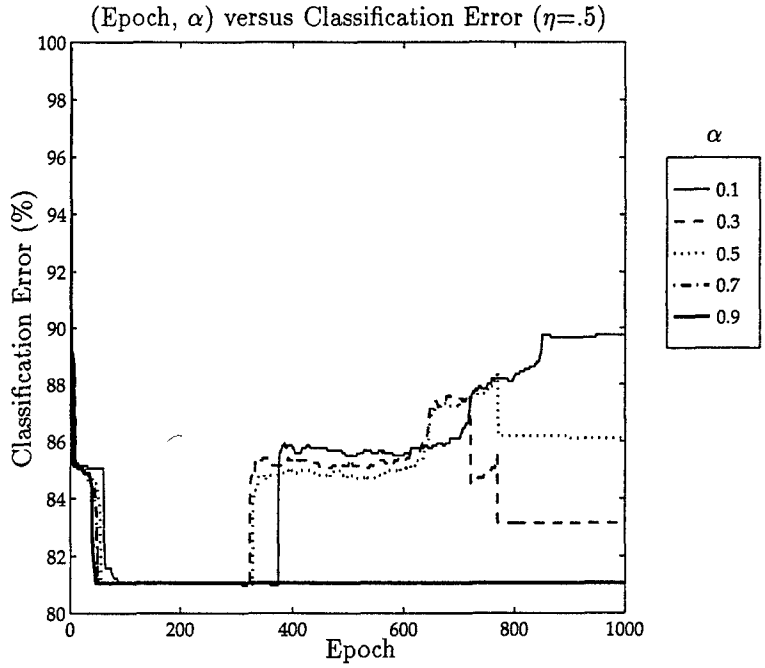



(a)

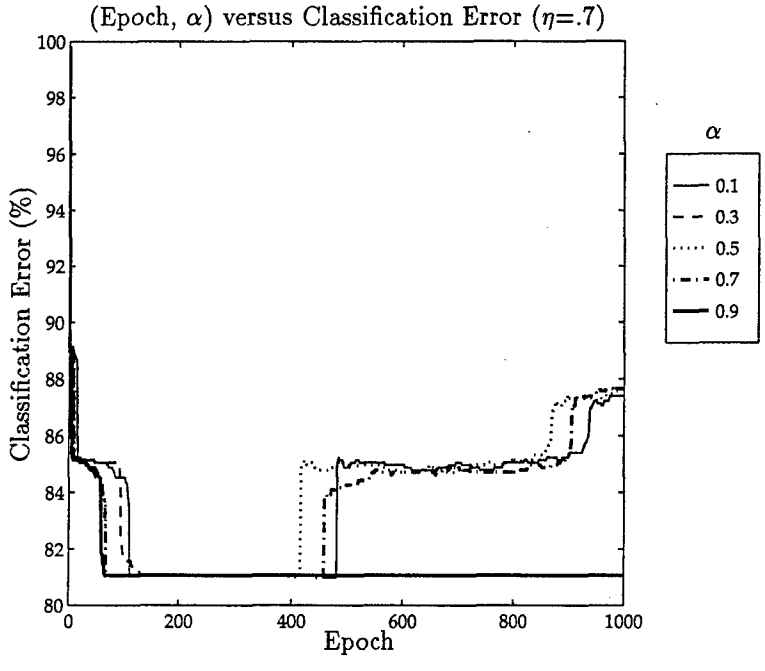



(b)

Figure D.4 $Actinide_1$ Fallside plots: classification error versus epoch for (a)$\eta$=.1 and (b)$\eta$=.3 over a range of $\alpha$ values.




(a)




(b)

Figure D.5 *Actinide*$_1$ Fallside plots: classification error versus epoch for (a)$\eta$=.5 and (b)$\eta$=.7 over a range of $\alpha$ values.

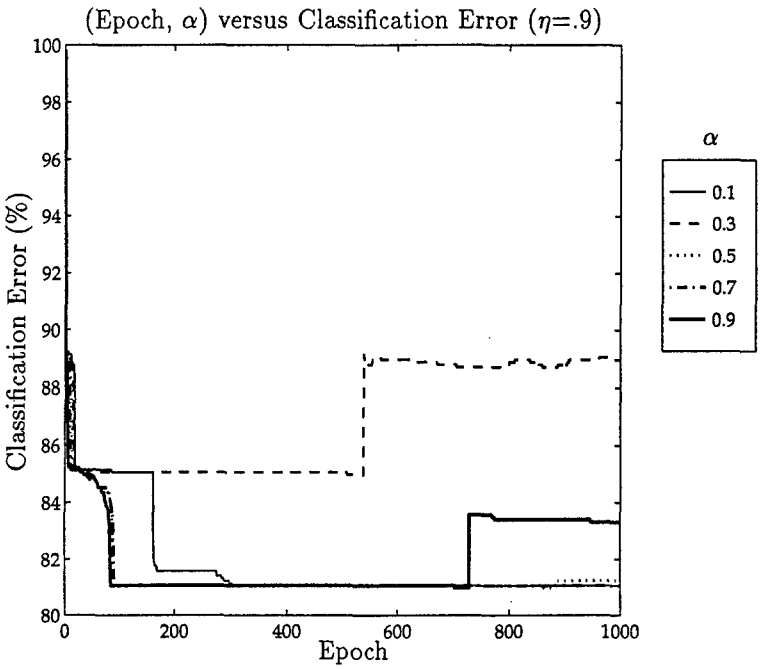
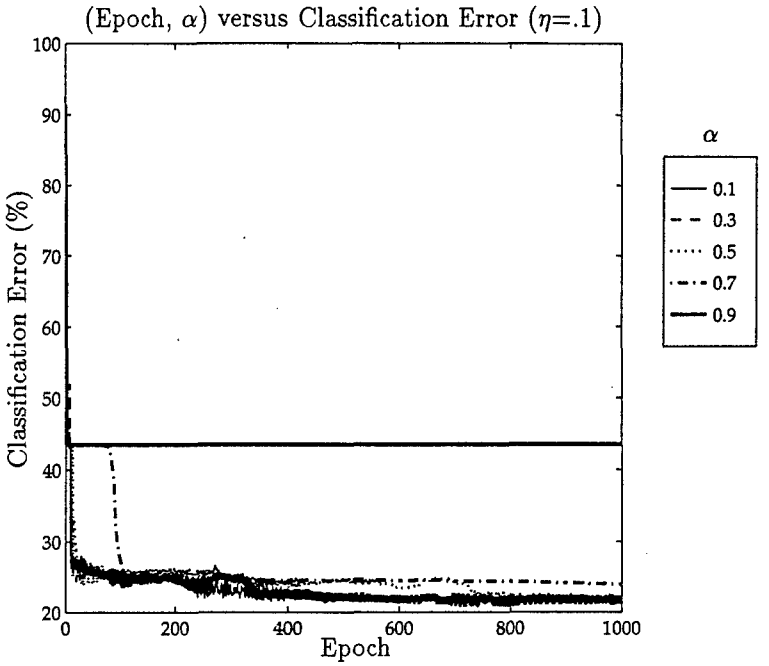
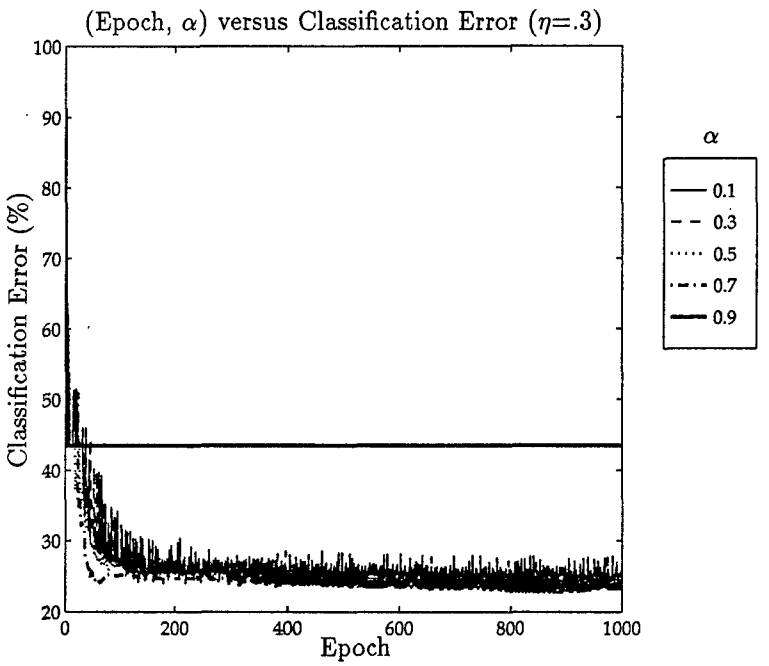



Figure D.6 *Actinide*$_1$ Fallside plots: classification error versus epoch for $\eta$=.9 over a range of $\alpha$ values.

## D.3 *Actinide*$_2$

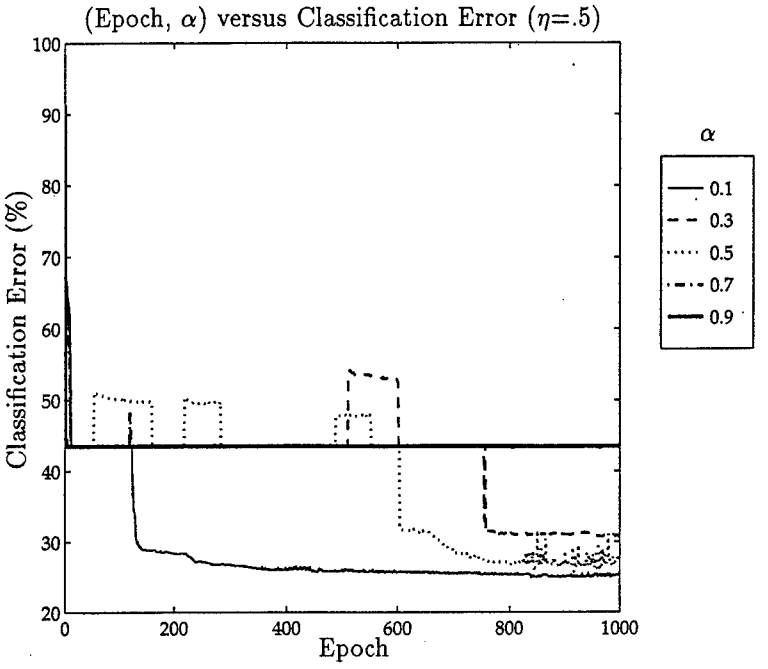



(a)

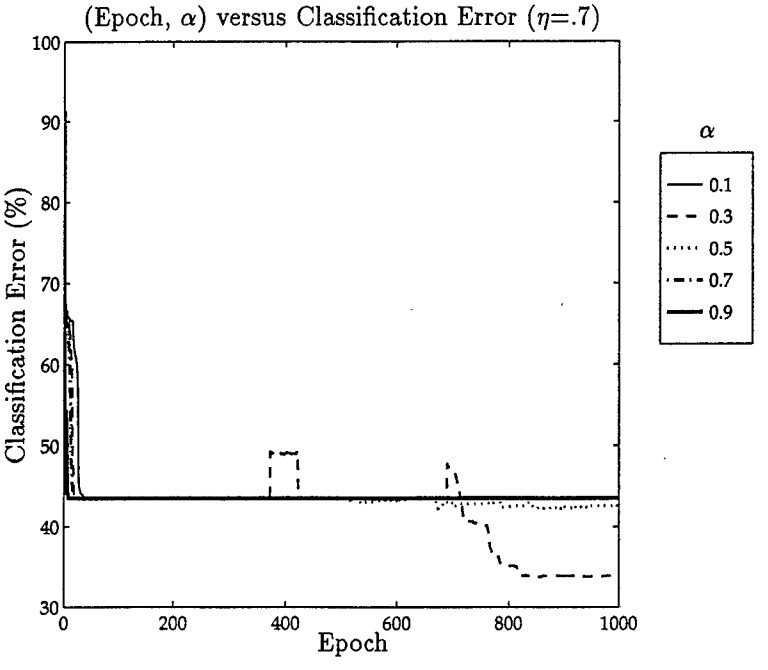



(b)

Figure D.7 *Actinide*$_2$ Fallside plots: classification error versus epoch for (a)$\eta$=.1 and (b)$\eta$=.3 over a range of $\alpha$ values.




(a)




(b)

Figure D.8 *Actinide*$_2$ Fallside plots: classification error versus epoch for (a)$\eta$=.5 and (b)$\eta$=.7 over a range of $\alpha$ values.

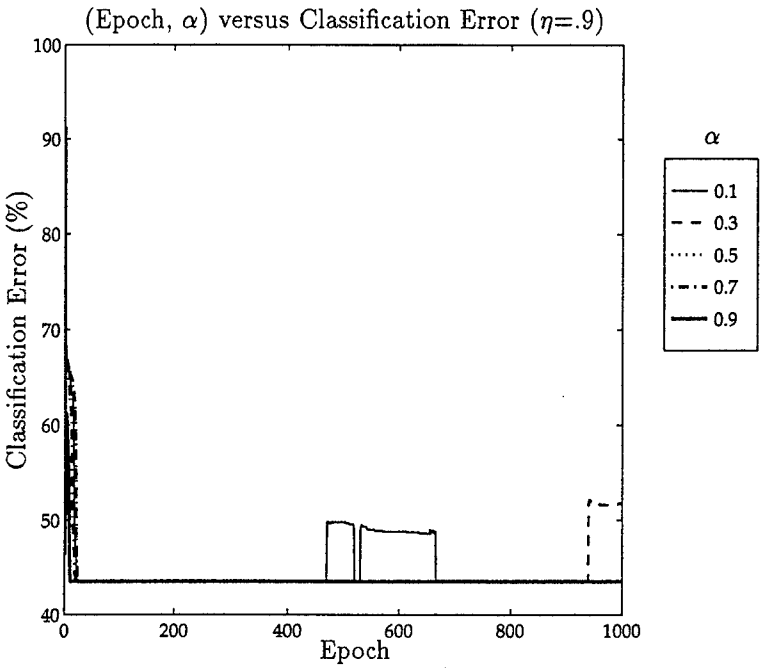



Figure D.9 *Actinide*$_2$ Fallside plots: classification error versus epoch for $\eta$=.9 over a range of $\alpha$ values.

*Appendix E. Actinide Classes*

| Class | Location | Class | Location |
|---|---|---|---|
| 1 | A-1 | 29 | M-11 |
| 2 | A-2 | 30 | M-12 |
| 3 | A-3 | 31 | M-13 |
| 4 | A-4 | 32 | M-14 |
| 5 | A-5 | 33 | M-15 |
| 6 | A-6 | 34 | M-16 |
| 7 | A-7 | 35 | M-17 |
| 8 | A-8 | 36 | M-18 |
| 9 | B | 37 | M-19 |
| 10 | C | 38 | M-2 |
| 11 | D-1 | 39 | M-20 |
| 12 | E | 40 | M-21 |
| 13 | F | 41 | M-22 |
| 14 | G | 42 | M-23 |
| 15 | H | 43 | M-24 |
| 16 | I-1 | 44 | M-25 |
| 17 | J | 45 | M-26 |
| 18 | J-1 | 46 | M-27 |
| 19 | J-2 | 47 | M-3 |
| 20 | J-3 | 48 | M-4 |
| 21 | K | 49 | M-5 |
| 22 | L | 50 | M-6 |
| 23 | L-1 | 51 | M-7 |
| 24 | L-2 | 52 | M-8 |
| 25 | L-3 | 53 | M-9 |
| 26 | M | 54 | M-N |
| 27 | M-1 | 55 | P-1 |
| 28 | M-10 | | |

Table E.1 Actinide Classes

*Bibliography*


1. Alpsan, D. "Are Modified Back-Propagation Algorithms Worth the Effort," *Proceedings from the International Joint Conference on Neural Networks* (1994).
2. Baum, Eric B. "What size net gives valid generalization," *Neural Computation* (1989).
3. Becker, S. and Y Le Cun. "Improving the convergence of back-propagation learning with second order methods," *Proceedings of the Connectionist Models Summer School* (1988).
4. Burns, J. A. and G. M. Whitesides. "Feed-forward neural networks in chemistry: Mathematical systems for classification and patter recognition," *Chemical Reviews*, *93*(8):2583–2601 (1993).
5. Carpenter, Gail A., et al. "Fuzzy ARTMAP: A Neural Network Architecture for Incremental Supervised Learning of Analog Multidimensional Maps," *IEEE Transactions on Neural Networks* (1992).
6. Carpenter, Gail A. and Stephen Grossberg. "Neural Dynamics of Category Learning and Recognition: Attention, Memory Consolidation, and Amnesia." *Brain Structure, Learning, and Memory* AAAS Symposium, edited by J. Davis, et al., 1986.
7. Chan, L.-W. and F. Fallside. "An adaptive training algorithm for back propagation networks," *Computer Speech and Language* (1987).
8. Cybenko, G. "Approximation by Superposition of Sigmoidal Functions," *Math. Control Signals Syst* (1989).
9. Duda, Richard O. and Peter E. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, 1973.
10. Foley, Donald H. "Considerations of Sample and Feature Size," *IEEE Transactions on Information Theory*, *IT-18*(5):618–626 (September 1972).
11. Fukunaga, Keinosuke and Donald M. Hummels. "Bayes Error Estimation Using Parzen and $k$-NN Procedures," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *PAMI-9* (1987).
12. Hopfield, J. J. "Neurons with Graded Response Have Collective Computational Properties Like Those of Two-State Neurons," *Proceedings of the National Academy of Sciences*, *81*:3088–3092 (1984).
13. Kabrisky, Matthew and Steven K. Rogers. "Sensory Communications." EENG 618 class notes, Air Force Institute of Technology, January 1996.
14. Kalman, Barry L. and Stan C. Kwasny. "Why Tanh: Choosing a Sigmoidal Function," *IEEE Proceedings of the International Joint Conference on Neural Networks* (1992).
15. Kohonen, Tuovo. *Self-Organization and Associative Memory*. Berlin: Springer-Verlag, 1984.
16. Leary, Dennis A. and Jerry J. Leary. *Principles of Instrumental Analysis* (fourth Edition). New York: Harcourt Brace College Publishers, 1992.

17. Lippmann, Richard P. "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine*, 4–22 (April 1987).

18. Martin, Curtis E. *Non-Parametric Bayes Error Estimation For UHRR Target Identification*. MS thesis, Air Force Institute of Technology, 1993.

19. Martin, Curtis E., et al. "Neural Network Bayes Error Estimation," *Proceedings of IEEE Conference on Neural Networks* (1994).

20. Parker, D. *Learning logic*. Technical Report, Office of Technology Licensing, Stanford, 1982.

21. Priddy, Kevin L. "Bayesian selection of important features for feedforward neural networks," *Neurocomputing* (1992).

22. Rathbun, Tom, et al. "Multi-layer Perceptron Iterative Construction Algorithm." Submitted to *Neural Computing.*

23. Rogers, Steven K., et al. *An Introduction to Biological and Artificial Neural Networks*. Air Force Institute of Technology, 1990.

24. Rosenblatt. *Principles of Neurodynamics*. New York, NY: Spartan Books, 1959.

25. Ruck, Dennis W. *Characterization of Multilayer Perceptrons and their Application to Multisensor Automatic Target Detection*. PhD dissertation, Air Force Institute of Technology, WPAFB, OH, December 1990.

26. Ruck, Dennis W. and Steven K. Rogers. "The Multilayer Perceptron as an Approximation to a Bayes Optimal Discriminant Function," *IEEE Transactions on Neural Networks*, *1*(2):296–298 (December 1990).

27. Ruck, Dennis W., et al. "Feature Selection Using a Multilayer Perceptron," *Journal of Neural Network Computing*, 40–48 (Fall 1990).

28. Rumelhart, David E. and McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, 1986.

29. Smagt, Patrick P. Van Der. "Minimisation Methods for Training Feedforward Neural Neural Networks," *Neural Networks*, *7*(1):1–11 (1994).

30. Steppe, Jean M. *Feature and Model Selection in Feedforward Neural Networks*. PhD dissertation, Air Force Institute of Technology, 1994.

31. Tou, Julius T. and Rafael C. Gonzalez. *Pattern Recognition Principals*. Reading, MA: Addison-Wesley Publishing Company, 1974.

32. Werbos, Paul. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD dissertation, Harvard University, 1974.

*Vita*

Jeffrey Leon Blackmon [redacted]. He graduated from Brandon High School in Brandon, Mississippi in May, 1989. He attended the University of Alabama and graduated *cum laude* with a Bachelor of Science in Electrical Engineering on 14 May , 1994. He was also commissioned on the same date as a Second Lieutenant in the United States Air Force. Lieutenant Blackmon's first active duty assignment was in Engineering Design, 90th Civil Engineers, F.E. Warren Air Force Base, Wyoming. In March of 1995, he was selected to attend the Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, to pursue a Master of Science degree in Engineering and Environmental Management. Upon completion of that assignment, Lieutenant Blackmon will be assigned to the 88th Air Base Wing Environmental Management Office, Wright-Patterson Air Force Base, Ohio.

[redacted]

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188



| 1. AGENCY USE ONLY *(Leave blank)* | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | November 1996 | Master's Thesis |

**4. TITLE AND SUBTITLE**
Neural Network Classification of Environmental Samples

**5. FUNDING NUMBERS**

**6. AUTHOR(S)**
1Lt Jeffrey L. Blackmon

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Air Force Institute of Technology
2950 P Street
WPAFB OH 45433-6583

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Air Force Technical Applications Center
1030 South Highway A1A
Patrick AFB, Fl 32925-3002

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution unlimited

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*
This research develops a general methodology for designing neural network classifiers for real-world environmental problems. This methodology is demonstrated through the design of a multi-layer perceptron to classify stainless steel and actinide samples. This research provides techniques for selecting architecture and training parameters, choosing the number of training epochs, reducing the feature sets, and evaluating classifier performance. For the stainless steel data, the feature set is reduced from 196 features to 54 features and the average training set error and test set error are .6% and 5.5%, respectively. The best results attained on the actinide data set are 24% training set error and 26% test set error. The actinide feature set is reduced from 18 feature to 9 features. The products of this effort are a concise methodology for developing neural network classifiers and the specifications for a multi-layer perceptron classifier for each of the data sets.

**14. SUBJECT TERMS**
environmental samples, neural networks, multi-layer perceptron

**15. NUMBER OF PAGES**
116

**16. PRICE CODE**

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |